

**USENIX**

**C++ CONFERENCE PROCEEDINGS**

**USENIX**

**C++**

**CONFERENCE  
PROCEEDINGS**

**San Francisco, California**

**April 9 - 11, 1990**

**SPRING**

**1990**

For additional copies of these proceedings contact

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710 USA

The price is \$28

Outside the U.S.A and Canada, please add  
\$18 per copy for postage (via air printed matter).

Past USENIX C++ Proceedings

C++ Conference	October 1988	Denver, CO	\$30.00
C++ Workshop	November 1987	Santa Fe, NM	\$30.00

Outside the U.S.A and Canada, please add  
\$20 per copy for postage (via air printed matter).

Copyright © 1990 by The USENIX Association  
All rights reserved.

This volume is published as a collective work.  
Rights to individual papers remain  
with the author or the author's employer.

UNIX is a registered trademark of AT&T.  
Other trademarks are noted in the text.



# Program and Table of Contents

## USENIX C++ Conference

San Francisco, CA

April 9-11 1990

**Monday, April 9**

*Tutorials*

9:00 - 5:00

**Tuesday, April 10**

**Buena Vista Ballroom**

*Opening Session*

9:00 - 10:00

Welcome

Jim Waldo, Hewlett Packard

Keynote Address

Adele Goldberg, ParcPlace Systems

*Break*

10:00 - 10:30

*Methodologies* Chair: Martin O'Riordon

10:30 - 12:00

Experiences with Object-Oriented Software Development ..... 1  
Nicholas Wybolt, Cadre Technologies, Inc.

Climbing the C++ Learning Tree ..... 11  
P. R. Jossman, E. N. Schiebel, J. C. Shank, AT&T Bell Laboratories

Design Criteria for C++ Libraries ..... 25  
Dr. James M. Coggins, University of North Carolina at Chapel Hill

*Lunch Break*

12:00 - 2:00

*Frameworks* Chair: Geoff Wyant

2:00 - 3:30

Reliable Distributed Programming in C++: The Arjuna Approach ..... 37  
Graham D. Parrington, University of Newcastle upon Tyne

Designing Portable Application Frameworks for C++ ..... 51  
Fergal Dearle, Glockenspiel

FOG/C++: A Fragmented Object Generator ..... 63  
Yvon Gourhant, Marc Shapiro, Institut National de Recherche en  
Informatique et en Automatique

*Break*

3:30 - 4:00

**Applications I Chair: Dr. James Coggins**

4:00 - 5:30

Object-Oriented Redesign Using C++: Experience with Parser Generators .....	75
Judith E. Grass, Chandra Kintala, Ravi Sethi, AT&T Bell Laboratories	
GPERF: A Perfect Hash Function Generator .....	87
Douglas C. Schmidt, University of California, Irvine	
C++ and Operating Systems Performance: A Case Study .....	103
Vincent F. Russo, Peter W. Madany, Roy H. Campbell University of Illinois at Urbana - Champaign	

**ET CETERA Chair: Jim Waldo**

8:00 - 10:00 p.m.

RIPE: An Object Oriented Robot Independent Programming Environment .....	115
David J. Miller and R. Charleene Lennox, Sandia National Laboratories	
SIC — A System for Stochastic Simulation in C++ .....	125
Bernd Kluth, Institute for Teleprocessing Aachen University of Technology	
A Type Parameterization Language for C++ .....	137
Richard Blinne, NCR Microelectronic Products Division	

---

**Wednesday, April 11**

**Buena Vista Ballroom**

**Invited Paper Chair: Andrew Koenig**

9:00 - 10:00

Exception Handling for C++ .....	149
Andrew Koenig and Bjarne Stroustrup, AT&T Bell Laboratories	

**Break**

10:00 - 10:30

**Applications II Chair: Roy Campbell**

10:30 - 12:00

Experiences in Writing a Distributed Particle Simulation Code in C++ ...	177
David W. Forslund, Charles Wingate, Peter Ford, J. Stephen Junkins, Jeffrey Jackson, Stephen C. Pope, Los Alamos National Laboratory	
The Conduit: A Communication Abstraction in C++ .....	191
Jonathan M. Zweig, Ralph E. Johnson University of Illinois at Urbana - Champaign	
Writing a Gateway in C++ .....	205
Preben Fisker Jensen, Peter Juhl, Jutland Telephone Co.	

**Lunch Break**

12:00 - 2:00

<i>Extensions</i>	<i>Chair: Peter Canning</i>	2:00 - 3:30
An Exception Handling Implementation for C++ .....	Michael D. Tiemann	215
Runtime Access to Type Information in C++ .....	John A. Interrante, Mark A. Linton, Stanford University	233
Extended C++ .....	Robert Seliger, Hewlett Packard Clinical Information Systems	241
<i>Break</i>		3:30 - 4:00
<i>Environments</i>	<i>Chair: Jim Waldo</i>	4:00 - 5:30
The C++ Information Abstractor .....	Judith E. Grass, Yih-Farn Chen, AT&T Bell Laboratories	265
Adding New Code to a Running C++ Program .....	Sean M. Dorward, Ravi Sethi, Jonathan E. Shopiro, AT&T Bell Laboratories	279
FIELD Support for C++ .....	Steven P. Reiss, Scott Meyers, Brown University	293

---

### *Advanced Topics Workshop*

Customization in C++ .....	Douglas Lea, SUNY	301
Does C++ Really Need Multiple Inheritance? .....	T. A. Cargill, Louisville, Colorado	315

## Program Committee

Jim Waldo	Apollo Computer, chair
Andy Koenig	AT&T
James Coggins	University of North Carolina, Chapel Hill
Martin O'Riordan	Microsoft
Geoff Wyant	Apollo Computer
Roy Campbell	University of Illinois, Urbana-Champaign
Peter Canning	Hewlett Packard

# Experiences With C++ and Object-Oriented Software Development

Nicholas Wybolt  
Providence Division  
Cadre Technologies Inc.  
Providence, Rhode Island

## ABSTRACT

*Object-oriented* is a very hot topic and buzzword both in academia and industry. There are object-oriented analysis and design techniques, object-oriented languages and databases, and so on. Many people see the letters "OO", attach a "G" to the front, and a "D" to the back and deem it to be "GOOD" — without much consideration for what it means in the software life cycle.

This paper discusses the on-going (3+ years) object-oriented re-design and re-implementation in C++ of a commercial CASE tool. Specifically, why an object-oriented approach was chosen and the implications and collective experiences of this approach. In addition to the anticipated benefits, much of what we experienced was unforeseen and unexpected.

## INTRODUCTION

It is believed that object-oriented (OO) approaches to software development hold great promise for increased programmer and organizational productivity. For over three years, the Providence Division of Cadre Technologies Inc. has been involved in the on-going re-design and re-implementation of a commercial CASE tool using an object-oriented approach and the C++ language. Although we have realized many benefits claimed by the OO mavens, we also experienced many unexpected surprises and problems.

This paper presents a brief history of our approach, how it changed our application, and describes some of the discoveries that we made about OO development and the C++ language, and what the ramifications of those discoveries have been. Finally, we present some observations and opinions on OO approaches and the C++ language and its associated technology.

## OVERVIEW OF THE APPLICATION

The application that is being migrated to an object-oriented implementation is Cadre Technologies Inc.'s Teamwork® CASE tool for systems analysis and design. Teamwork has been available since June of 1985 and runs on nine workstation platforms.

Teamwork is a *design automation tool*. It automates a number of methodologies known as *Structured Analysis* [DeMarco 1978], *Real-Time Analysis* [Hatley 1987],

*Information Modelling* [Chen 1976], *Structured Design* [Page-Jones 1980], *Ada Structure Graphs* [Buhr 1984], and *Object-Oriented Analysis* [Shlaer 1988]. Teamwork supports multiple users working simultaneously on the same project or on multiple projects in a Local Area Network. All of this is coupled with an open and extensible environment accessed through a public tool interface.

## Teamwork Architecture

The Teamwork architecture is very much a classic client-server model of computation. There is a *display manager* component that implements the user interface. There is a *database* in which analysis and design models reside and a *data controller server* that creates, modifies, and examines objects in the database. The clients consists of one or more *application clients* (or editors) that interact with the display manager and the database server. Finally, the display manager and the application clients all run within a *desktop* metaphor on the user's workstation.

## User's View of the Teamwork Architecture

The user's view of the Teamwork architecture is through the user interface which is identical on all workstation platforms. The user interface consists of: pop-up menus, multiple, independent windows, window icons, cursors in the shapes of objects that can be sized, user-defined groups of objects, undo/redo operations, and so on.

The user perceives Teamwork as being *object-based* in the sense that all Teamwork objects and user interface graphics idioms are treated and manipulated as single and composite



objects. Specifically, there are multiple windows that have a one-to-one correspondence with objects in the database. The database itself is a network of textual and graphical objects. Each of the objects in the database is typed. That is, there is a different syntax-directed editor for each object that incorporates the analysis and design model building rules. Figure 1 illustrates a sample Teamwork window.

The Teamwork window is manipulated as an object and consists of other objects. The data flow diagram displayed in the window is an object as are the individual flows, bubble, flow names (data dictionary entries), and terminators. Similarly, the pop-up menu is an object.

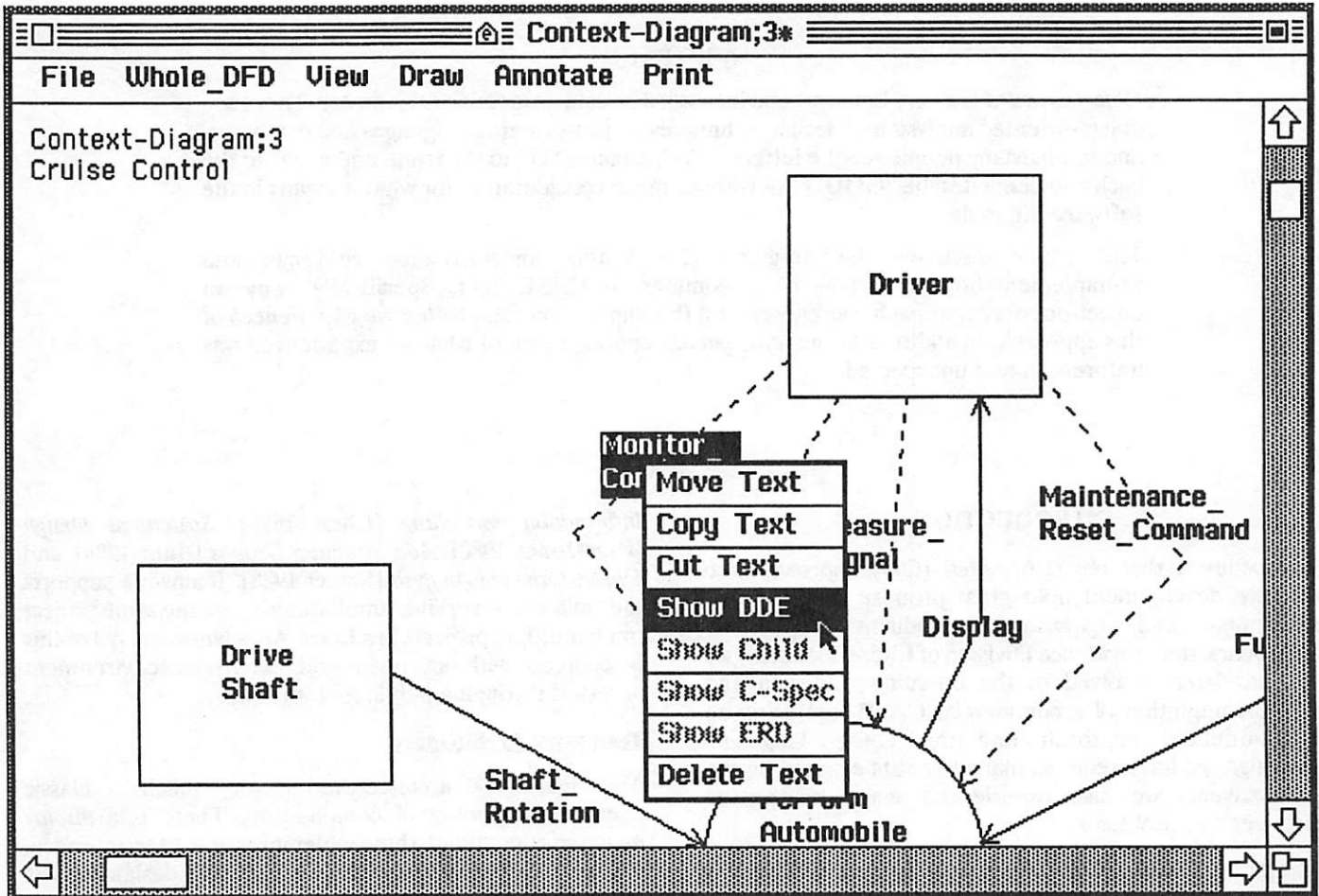


Figure 1. Sample Teamwork Window With Pop-Up Menu

The inherent object properties of Teamwork was a primary motivator for choosing an object-oriented re-design and re-implementation approach.

### MOTIVATION FOR CHOOSING AN OBJECT-ORIENTED APPROACH

Providing new functional capabilities in Teamwork primarily consists of adding a new editor that automates a structured methodology. The first Teamwork editor was for Structured Analysis. Subsequent editors added automation for Real-Time Analysis, Structured Design, and so on.

### The Way We Were

Typically, new editors (i.e., new function) were added by *cloning* existing code and modifying it to provide application-specific functions. At the time, this seemed to be the right way to proceed as time-to-market considerations were important and cloning is a relatively quick and easy process.

### Why A Change Was Necessary

This approach was not without problems, however. Specifically, as the product grew in size and function we experienced problems with the size of the source code and

the executables, maintenance became a real headache, and consistency among the editors began to diverge.

Each new editor typically brought with it 25,000 to 67,000 new lines of C code. Dialog boxes, common to each editor, had to be manually copied from editor-to-editor. In addition, we unfortunately propagated bugs through the cloning process. Consequently, bug fixes had to be made in many places.

Problems with consistency arose because each new editor added new, application-specific functions, "improved" old functions, or added new "common" functions. No protocols, in particular, were followed when these new functions were added or old functions were changed. Consequently, new or improved features/functions were not always retro-fitted into the existing editors.

### Alternative Approaches

In mid 1986, The pain level associated with maintenance, in particular, reached a threshold where we decided to investigate the feasibility of a general code clean-up and the potential re-design and re-implementation of key portions of Teamwork.

In considering this process, we identified three alternative approaches:

1. Common reusable code using shared data structures and algorithms,
2. Responsible engineers assigned to specific products functions/editors, or
3. Re-implement editors using object-oriented design/implementation technology.

An approach based on common reusable code brings with it the potential for huge unions and many switch statements. Furthermore, adding something new could require a massive editing job.

Responsible engineers provide a transient solution in that the knowledge is lost if an engineers leaves. Correcting common bugs would have to be accomplished by many people and is further complicated through the need for coordinated communications among all parties concerned.

The object-oriented approach seemed to be the most elegant. It would allow us to create shared, reusable objects and to collect data and its associated functions. Changes would be localized and we would benefit from a reduction in code size.

Teamwork also seemed to be a "natural" for an object-oriented approach. It consists of many common objects such as windows, menus, lines, rectangles, circles, etc. Applied to these common objects are a whole list of common functions: move, copy, re-size, delete, clip, bind, and so on.

### Why C++?

Our choice to use the C++ language was not based on an exhaustive survey of OO languages available at the time. Nor did we consider the merits of a hybrid language versus a pure OO language. Rather, it was based on the pragmatics of our product offering: Teamwork has to run on nine different workstation platforms and the only OO language technology available to us was C++. Plain and simple.

We chose to use *Designer C++*<sup>™</sup>, a product marketed and sold by Oasys. Originally, we worked with the AT&T C++ translator; however, we would have owned the porting and support responsibilities for the AT&T product which we felt was not a viable course to follow.

*Designer C++* is a translator: It translates C++ source code into C source code. We perceived this as an additional benefit in that we believed that we would be able to enjoy some degree of development tool sharing between C and C++ source code. We also assumed that the learning curve from C to C++ would not be very significant.

### DESIGN AND IMPLEMENTATION APPROACH

The conceptual architecture of the Teamwork product, specifically the non-OO editors, is illustrated in Figure 2.

Input, supplied by the user, is captured by the editor and dispatched to the appropriate server for further processing. If the user wishes to view a new structure chart, for instance, a request is sent to the data controller server to retrieve the appropriate object. If the user wishes to icon a window, that request is sent to the display manager.

The Teamwork *object*, how it is *edited*, and how it *looks*, is all integrated into the same application-specific code and structure. That is, the presentation and editing of the objects are intricately woven together.

The conceptual architecture of the OO editors is illustrated in Figure 3.

The architecture of the editors has been changed such that the user interface component has been divorced from the editor. In this architecture, user input is captured by the user interface component and is then dispatched either to the editor or to the display manager for further processing.

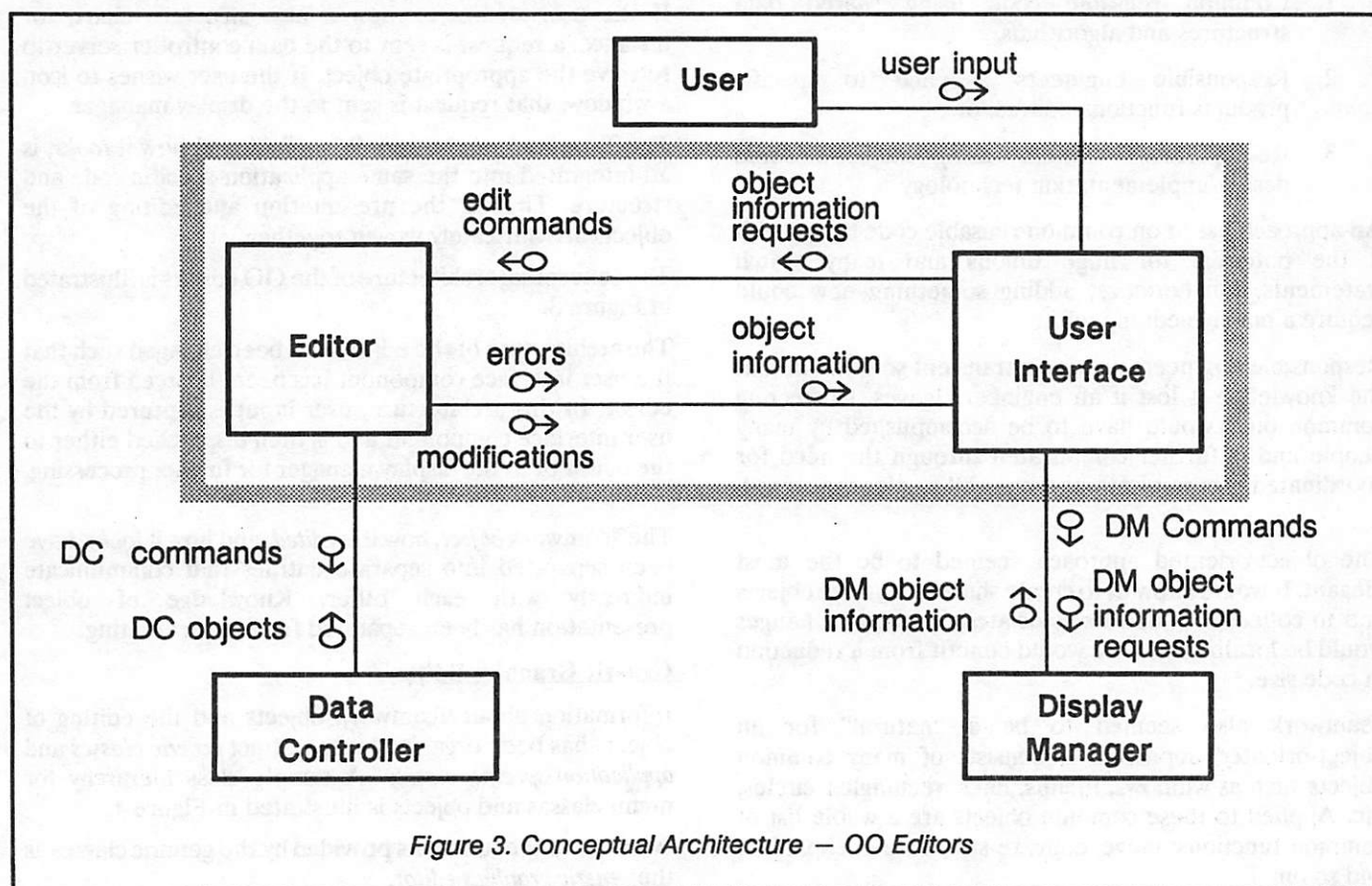
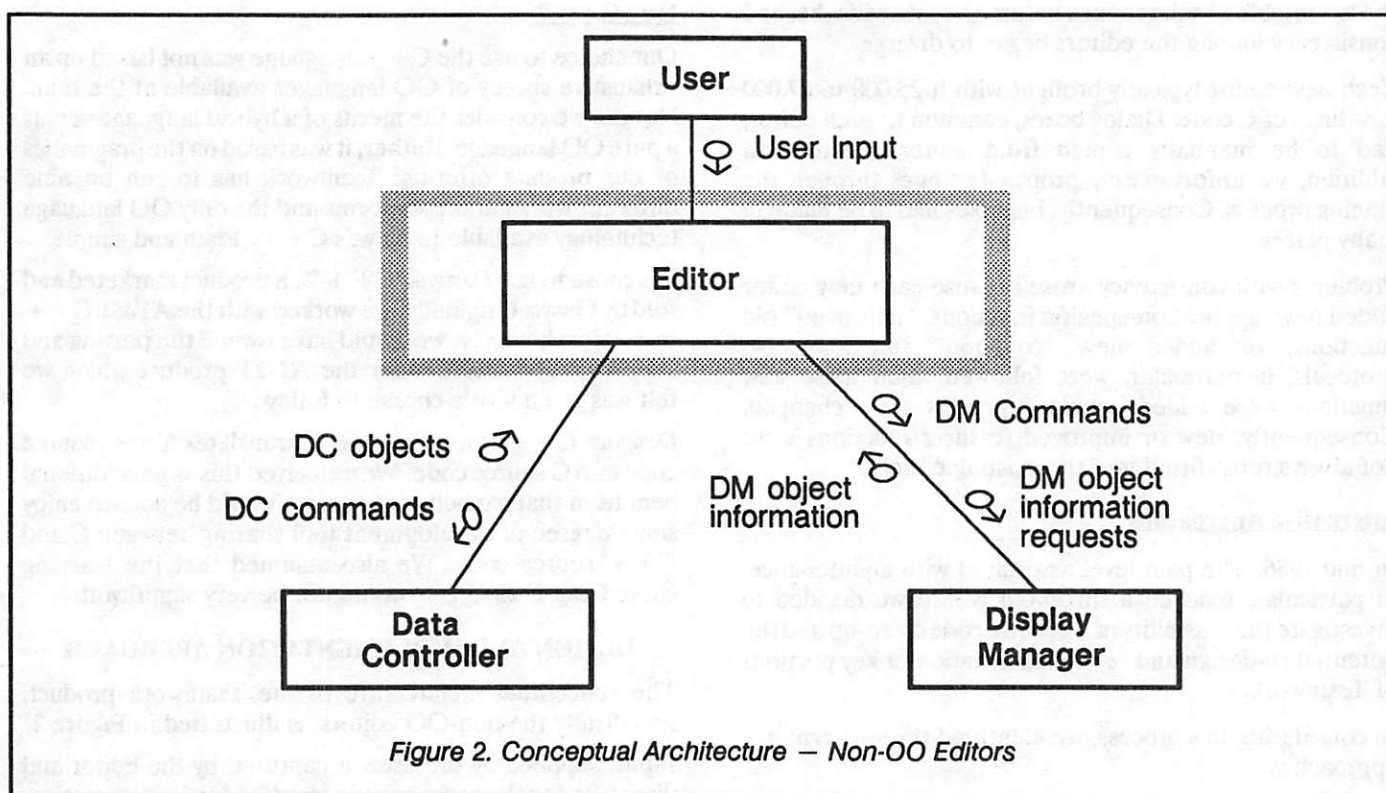
The Teamwork *object*, how it is *edited*, and how it *looks*, have been separated into separate entities that communicate indirectly with each other. Knowledge of object presentation has been separated from object editing.

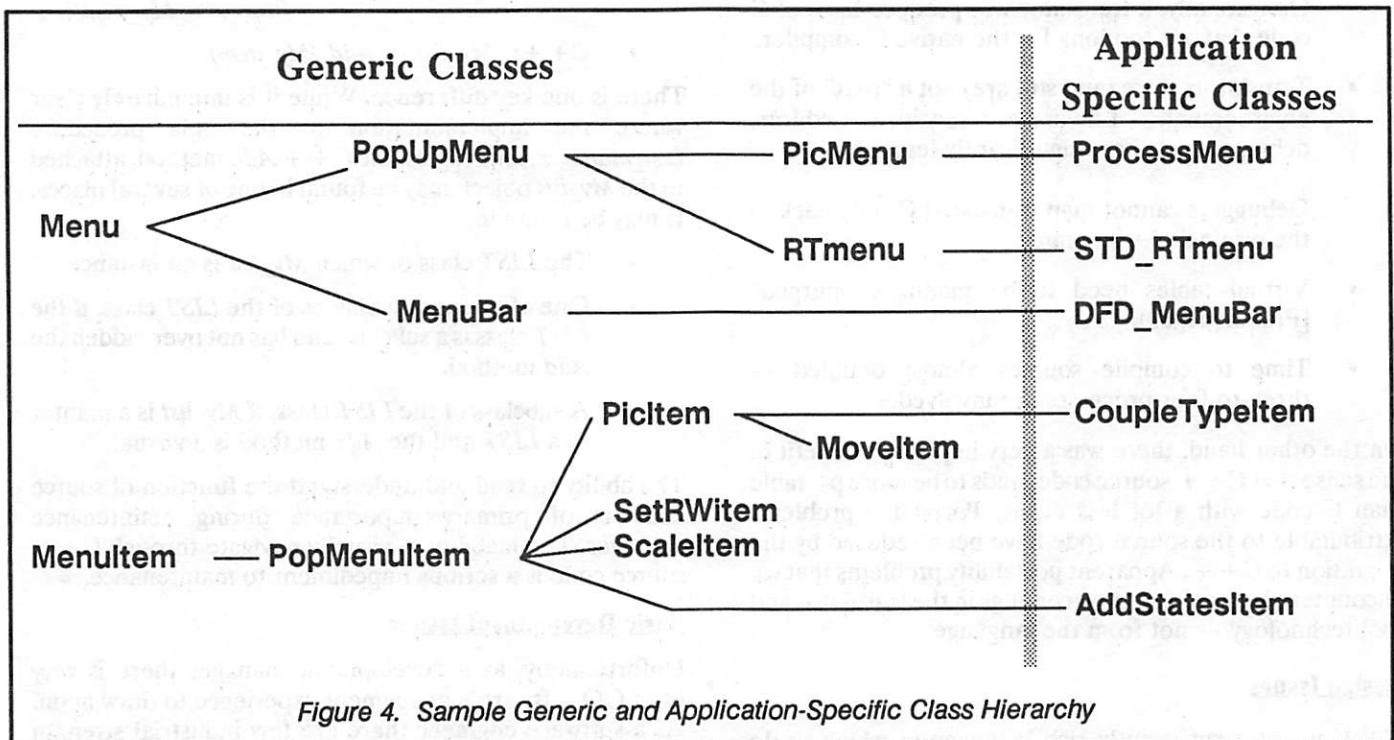
### Generic Graphics Editor

Information about Teamwork objects and the editing of objects has been organized into distinct *generic classes* and *application-specific classes*. A sample class hierarchy for menu classes and objects is illustrated in Figure 4.

We refer to the functions provided by the generic classes as the *generic graphics editor*.







Therefore, whenever a new editor is to be added, the software engineer simply adds application-specific classes that build on the generic classes (i.e., sub-classes to the existing classes).

### Some Results

One of the motivations for the Teamwork re-implementation was to reduce product size. Each new editor adds 3,000 to 10,000 lines of C++ code, instead of 25,000 to 67,000 lines of C code in the non-OO implementation.

Maintenance has been simplified. Dialog boxes and other common user interface idioms are defined in one place, rather than being spread across editors. Similarly, bug fixing is easier. A fix is made once and is propagated to all relevant editors.

Problems with consistency are managed because old functions are now shared and "improvements" are made once. New common functions are easily added to both new and old editors concurrently. We do not have to locate all references to a particular function, for instance.

Each of these benefits can be attributed to the division of the editing functions into generic and application-specific classes (i.e., the creation of the generic graphics editor).

### **THINGS WE FOUND ALONG THE WAY**

The following sections describe some of the more salient discoveries that we made during the re-design and re-implementation of the Teamwork editors.

#### Product Issues

Once the generic graphics editor was in place, adding new editors (graphic idioms, notation, etc.) became much easier. In addition, it simplified the product's architecture by decoupling syntax and semantics from presentation issues.

This simplification will be important to us in the future as we migrate Teamwork away from its own display management environment to an X-Window Manager™ environment, for instance.

#### C++ Technology Issues

Many of our early experiences with C++, specifically the translators, was not very positive. Much of this can be attributed to our being on the "bleeding edge" of the technology. Some unforeseen problems that we discovered were:

- The same version of the translator was not available across all platforms.
- Translator support was better for some platforms than for others.
- Translator output was not always compatible with host C compilers, librarians, linkers, and binders.

Occasionally, a translator will produce lines of C code that are too long for the native C compiler.

- Translators were (and still are) not a “part” of the environment. Language sensitive editors, debuggers, etc., are significantly less useful.

Debuggers cannot map translated C code back to the original C++ source.

- Virtual tables need to be manually “purged” [Fleming 1989].
- Time to compile sources almost doubled — three-to-four processes are involved.

On the other hand, there was a very important benefit in the sense that C++ source code tends to be more portable than C code with a lot less effort. Portability problems attributable to the source code have been reduced by the migration to C++. Apparent portability problems that we encountered arose from shortcomings in the translator and tool technology — not from the language.

### Design Issues

C++ is not a sufficiently rich language in which to do design directly. When we started (1986), there were no existing object-oriented design (OOD) methods or CASE tools to support OOD. Consequently, we had to improvise by building on, and extending existing methods and tools. Some of our findings are described later in this paper.

### Reuse

Obviously, we have derived benefits from the OO concept of reuse. However, reuse has an interesting attribute that we did not initially foresee.

Specifically, the unit of reuse is not the same as in C, for instance. In C, if you wish to make use of a mathematical function, you simply include the appropriate header file(s) and invoke the function. In C++, you must reuse the whole class hierarchy, not just the object.

Class hierarchies are not modular. You cannot just select the object(s) that you are interested in using [Guthery 1989]. This concept goes against “traditional” concepts of program modularity [Wirth 1974], and takes some getting used to.

### Navigation and Maintenance

Visual navigation through C++ source code has proved to be more difficult than with other languages with which we have experience, most notably C and Ada. For example, the following two subprogram/method invocations are equivalent in effect (i.e., adding an item to a list) [Wild 1989]:

- **Ada:** *List\_manager.Add (List => My\_list, Item => My\_item);*
- **C++:** *My\_list->Add (My\_item);*

There is one key difference: While it is immediately clear where the implementation of the Ada procedure *List\_manager.Add* resides, the C++ *Add* method, attached to the *My\_list* object, may be found in one of several places. It may be found in:

- The *LIST* class of which *My\_list* is an instance
- One of the parent classes of the *LIST* class, if the *LIST* class is a subclass and has not over-ridden the *Add* method.
- A subclass of the *LIST* class, if *My\_list* is a pointer to a *LIST* and the *Add* method is a virtual.

The ability to read and understand the function of source code is of primary importance during maintenance activities. The inability to visually navigate through C++ source code is a serious impediment to maintenance.

### Basic Development Issues

Unfortunately, as a development manager there is very little OO software management experience to draw upon. As a software engineer there are few industrial strength tools for C++ development available.

The unfamiliar aspects of class reuse, as far as modularity is concerned, run counter to many prevailing development practices and experiences.

Basic “uses” relationships in configuration management become more complicated. In addition to the “uses” relationship, developers and configuration management systems must contend with “inherits” relationships.

The expectations of management, as far as tangible progress is concerned, using an object-oriented approach must be set differently. There is a great deal more “up-front” work involved with respect to setting up class hierarchies and methods. The notion of “it works” may only mean that a particular message has gotten through to the correct method in the class hierarchy.

### **OBSERVATIONS AND OPINIONS**

The following represents some of our collective observations and opinions concerning software development using object-oriented technology. They are certainly not universal in academia, industry, nor within Cadre. They are offered as insights based on our experiences.

### Class Inheritance Paradigm

A question that arises is whether the class inheritance paradigm supports and preserves the benefits of encapsulation. Unfortunately, the answer seems to be “no”, because cohesion is sacrificed.



Instead of gathering functions together, it scatters them among parent and child classes. Instead of isolating the effect of changes, it creates dependencies between a class' methods, its super-class' methods, and its sub-class' methods.

The use of inheritance does not necessarily isolate where functions can be found, nor does it localize their effect.

Avoiding classes is not the solution to this problem. Rather, it is the interdependencies between classes that must be watched and minimized. This is hard to do — while creating a hierarchy of tightly inter-operating classes, a structure of dependencies is being (unconsciously) created.

The dependency structure of a system deserves at least as much attention as do its reuse characteristics. The maintainability of a system seems to be inversely related to the number of dependencies between classes/methods.

### Up-Front Analysis

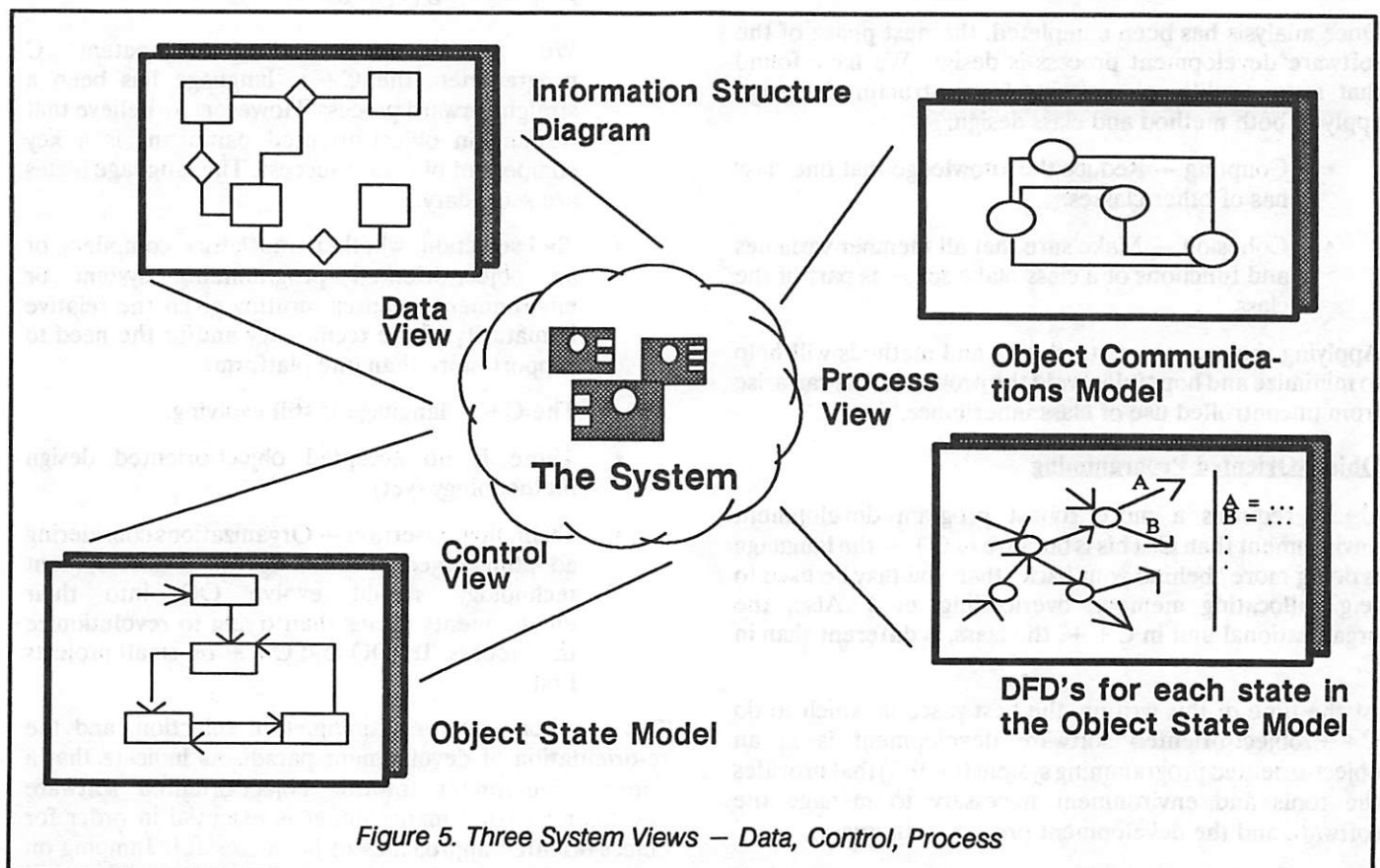
When we first started this project, we did not bother performing any analysis activities. If we had a question about how the system should behave, we simply brought Teamwork up — “Ask the computer!”

However, over the course of the past year, we've come to believe that performing some level of object-oriented analysis (OOA) is critical. It lets the software engineer discover the objects (data) of the system being developed, their attributes, and relationships. The behavior of each object can be defined, and a description of how events (messages) are propagated and handled (methods) can be derived.

The result is that the basic structure of class hierarchies can be determined and the persistence of the objects in the system can be determined as well. A determination of object persistence enables the software engineer to accurately convey the life cycle of an object.

The OOA approach that we have used successfully is based on a re-orientation and extension of traditional notations for data flow diagrams (DFD's), state transition diagrams (STD's), and entity-relationship diagrams (ERD's) [Shlaer 1988]. In the OOA approach we first model the data in the system using ERD's — producing an information model, describe the life cycles of objects using STD's — producing an object state model, and define the objects and their life cycles, a process view, using DFD's — producing an object communications model.

Figure 5 illustrates these three system views.



Some would argue that "traditional" functional decomposition methodologies are not suited for object-oriented systems. However, whether analyzing a system using functional or object-oriented techniques, three things remain common:

1. What is being done — the algorithm,
2. What it is being done to — the data, and
3. When it is being done — the control.

The difference between functional decomposition and an object-oriented approach is simply the order in which each of these is applied. Therefore, the analysis technique which we use is even more appealing in that it makes use of these "traditional" and familiar notations, rather than introducing a new notation(s).

Using ERD's to represent classes has an additional benefit in that an ERD editor can be used effectively as a class browser. We have done some initial prototyping whereby an ERD editor has been coupled with a code management system such that software engineers can navigate from the entities in the ERD's to the C++ source code. The early results have been promising.

### Design

Once analysis has been completed, the next phase of the software development process is design. We have found that some traditional concepts from Structured Design apply to both method and class design:

- Coupling — Reduce the knowledge that one class has of other classes.
- Cohesion — Make sure that all member variables and functions of a class make sense as part of the class.

Applying these concepts to classes and methods will help to minimize and hopefully avoid the problems that can arise from uncontrolled use of class inheritance.

### Object-Oriented Programming

C++ requires a more robust program development environment than C. This is because in C++ the language is doing more "behind your back" than you may be used to (e.g., allocating memory, overloading, etc.). Also, the organizational unit in C++, the class, is different than in C.

At the time of this writing, the best place in which to do C++/object-oriented software development is in an object-oriented programming system (OOPS) that provides the tools and environment necessary to manage the software and the development process. Otherwise,

- Managing class hierarchies and libraries is difficult without browsing tools.

- Editing methods is complicated because the method may reside in one of many places within the class hierarchy.
- Debugging is difficult because of the difficulty mapping translated C source to C++ source.
- The toolsets are still rather immature. Those that exist fit orthogonally into existing environments.

The expectations and excitement over C++ and object-oriented programming is about one year ahead of the technology curve.

### SUMMARY

*Object-oriented* is a very hot topic and buzzword both in academia and industry. We have presented some insights on the "OO bandwagon" based on the on-going object-oriented re-design and re-implementation in C++ of a commercial CASE tool. Our experiences have led us to the following conclusions, opinions, and warnings concerning object-oriented software development and, in particular, C++:

- Training is essential. This is particularly true of educating software engineers in object-oriented programming paradigms.

We found that teaching competent C programmers the C++ language has been a straightforward process. However, we believe that training in object-oriented paradigms is a key component of future success. The language issues are secondary.

- Tool selection, whether translators, compilers, or an object-oriented programming system or environment requires scrutiny given the relative immaturity of the technology and/or the need to support more than one platform.
- The C++ language is still evolving.
- There is no accepted object-oriented design methodology (yet).
- Technology insertion — Organizations considering adopting object-oriented software development technology should evolve OO into their environments rather than trying to revolutionize the process. Try OO and C++ on small projects first.

The requirements for training, tool selection, and the re-orientation of development paradigms indicate that a strong commitment towards object-oriented software development from management is essential in order for object-oriented approaches to be successful. Jumping on the "OO bandwagon" without this commitment is a recipe for failure.

## ACKNOWLEDGEMENTS

The author is indebted to the following people who reviewed and made many useful comments on early drafts of this paper: Jim Amsden, Fred Barrett, Read Fleming, Dave Fortin, Jacquy Harkness, Don Heitzmann, and Fred Wild. David Fortin collected and described the information and product architecture presented in the *Design and Implementation Approach* section. Alan Hecht and David Taylor described the application of Structured Design concepts to object-oriented design.

## REFERENCES

- Buhr, R. J. A. 1984. *System Design with Ada*, Prentice-Hall Inc., Englewood Cliffs, NJ.
- Chen, P. 1976. "The Entity-Relationship Model — Toward a Unified View of Data," *ACM Transactions on Database Systems*, pp. 9-36.
- DeMarco, T. 1978. *Structured Analysis and System Specification*, Yourdon Press, NY.
- Guthery, S. 1989. "Are The Emperor's New Clothes Object-Oriented?", *Dr. Dobb's Journal*, 14,12, (December 1989) pp. 80-86.
- Fleming, R. 1989. *Managing C++ Virtual Tables*, Unpublished manuscript, Cadre Technologies Inc., Providence, RI.
- Hatley, D. J. and Pirbhai, I. A. 1987. *Strategies for Real-Time System Specification*, Dorset House, NY.
- Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design*, Yourdon Press, NY.
- Shlaer, S. and Mellor, S. J. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*, Yourdon Press, NJ.
- Wild, F. H. 1989. *A Comparison of Experiences With The Maintenance of Object-Oriented Systems: Ada vs. C++*, Unpublished manuscript, Cadre Technologies Inc., Providence, RI.
- Wirth, N. 1974. "On the Composition of Well-Structured Programs," *ACM Computing Surveys*, 6,4, (December 1974) pp. 247-259.





# Climbing the C++ Learning Curve

Paul R. Jossman  
Edward N. Schiebel  
Jere C. Shank

AT&T Bell Laboratories

## Abstract

Learning to use the object-oriented features of C++ requires climbing a long learning curve. This paper presents some lessons we learned while working on our first project in C++ in an attempt to make the climb a little easier for other new-comers.

## 1. Introduction

Almost three years ago we began working on a set of computer-aided design tools and were considering using C++. In an effort to learn more about programming in this new language, we attended the first USENIX C++ Workshop. One of the first things we discovered was object-oriented programming in C++ has a learning curve of about one year! We had deadlines to meet, could we possibly work in this new style, this new language, and finish? We found Bjarne Stroustrup and asked him whether C++ was right for us. Bjarne's (completely unbiased) opinion was yes, the learning curve was long, but in a year we would have progressed farther than if we had worked in "old" C. Now, three years, four prototypes and two released applications later we have learned a lot.

In this paper we discuss some lessons we have learned, some revelations we have had, and some unresolved issues. Section 2, *A Neophyte's Guide to C++* provides helpful hints on selected language related topics. Section 3, *Object-Oriented Design*, discusses some general factors to consider when designing your application. Section 4, *Sales Tax or Inheritance Tax, Either Way You Pay*, offers opinions on the own versus inherit war. Section 5, *Functionoids and Glue Objects*, describes techniques used to keep major pieces of an application independent and reusable. Section 6 contains our conclusions.

## 2. A Neophyte's Guide to C++

### 2.1 Helpful Hints

**References.** Reference types, seemingly simple, turned out to be difficult for us to master. We vacillated between "pointers have no business in object oriented programs, let's only use references" and "reference semantics are peculiar, who needs the hassle." We wound up using references with varying degrees of success in three places: as function arguments, as function return values, and as member data.

The most useful application of references is as function arguments. Large objects may be passed to functions with low overhead and accessed with a more natural syntax than pointer dereferencing. Used with `const`, you get the same data protection as pass by value with the efficiency of pass by reference.

Returning a reference to member data from a member function permits the function result to be used as an lvalue:

```
class Foo {
public:
    Foo() : i(0){}
    int& g() {return i;}
private:
    int i;
};
```

```
main()
{
    Foo f;
    f.g() = 10;    // f::i = 10
}
```

but this use of returning references trades convenience for a violation of data hiding principles <sup>[1] [2]</sup>.

Returning a reference from a function (\*this, member data, or the left operand of a binary operator) enables function calls to be cascaded.

```
class Foo {
public:
    Foo() : i(0) {}
    Foo& increment() {++i; return *this;}
private:
    int i;
};

main()
{
    Foo f;
    f.increment().increment().increment();    // f::i == 3
}
```

Cascading function calls can be convenient (as with I/O operators), but beware that error conditions may occur between the cascaded calls.

We also tried using references as member data, but with mixed results. On the plus side, it guarantees the references will be initialized at construction (the compiler will flag a missed initialization as an error), and that the reference *always* refers to the same object.

The use of references as member data does pose a hidden danger. The compiler provides a default assignment operator (e.g. `X::operator=(const X&)`) that performs recursive member-wise assignment. For pointer members, a left-hand-side pointer's value is changed and it will point to a different object, i.e. the object pointed to by the right-hand-side pointer member. For references, a left-hand-side reference can never be changed to "refer" to a different object, instead the right-hand-side object will assign its members to the left-hand-side's referenced members. For example:

```
#include <iostream.h>
class Foo {
public:
    Foo(int i) : ir(i) {}
private:
    int& ir;
};

main()
{
    int i1 = 5;
    int i2 = 7;
    Foo f1(i1);
    Foo f2(i2);

    cout << "i1 = " << i1 << endl;    // outputs: i1 = 5
    cout << "i2 = " << i2 << endl;    // outputs: i2 = 7
    f1 = f2;
}
```

```
    cout << "i1 = " << i1 << endl;    // outputs: i1 = 71
}
```

Careful programming can enforce initialization and constant reference on pointer types, avoiding the dangers inherent in assignment.

For further discussions on references, see [3] and [4].

**Constructors and Destructors.** Constructors and destructors look like member functions, are coded like member functions, but are distinctly different.

Unlike other functions, constructors and destructors are not explicitly called. The language generates hidden calls to constructors and destructors when variables enter and leave scope and when temporaries are created. Not only are the constructors you provide called implicitly, but if you don't provide an `X::X(const X&)` constructor, the compiler generates one for you. By definition, this constructor does member-wise recursive assignment and may cause problems, especially if `X` allocates members on the free store.

Another difference is constructors are not inherited. For example:

```
class Base {
public:
    Base();
};
class Derived : public Base {
public:
    Derived(int);
};
main()
{
    // error: argument 1 of type int expected for Derived::Derived()
    Derived d;
}
```

The error results because `Base::Base()` is *not* inherited by `Derived`.

On the other hand, destructors *are* inherited, but *must* be declared as virtual in the base class if you are exploiting polymorphism and expect a derived class' destructor to be called. In this context, destructors behave identically to other member functions.

**Constant Member Functions.** A new feature in Release 2.0, there are two interpretations to the meaning of constant member function: the object's member data may not be modified during the function call (the compiler's view); and the object's external view through its public interface will not change during the function call (a more pragmatic view).

The compiler's approach seems understandable, since constant memory is the only "constness" the compiler can verify, but this interpretation limits the usefulness of `const`. For example, one reason we violated the compiler's definition was to cache computed values. Our `Matrix` class cached a state variable indicating whether it was the identity matrix:

1. Currently, `cfront 2.00` contains a bug that treats reference members as if they were pointers in this situation, i.e. what they refer to is changed, not their internal state. Thus, this line will actually output `i1 = 5`.

```

class Matrix {
public:
    // ...
    enum Result {unknown = -1, no, yes};
    Result isIdentity() const;
private:
    Result isI; // Is this matrix the identity matrix?
};
Result
Matrix::isIdentity() const
{
    if(isI == unknown){
        // compute state of this matrix

        isI = computed_state_of_matrix; // error:
        // assignment to member Matrix::isI of const struct Matrix
    }
    return isI;
}

```

Our opinion is, that from a user's point of view, `isIdentity()` does not change its `Matrix`, and declaring it `const` is reasonable. Unfortunately, if you wish to use this more pragmatic interpretation of `const`, you have to convince the compiler that you are still playing by its rules.

In a constant member function, this is of type `const X*`. To alter any member data you must either cast away this "constness", or use a level of indirection:

```

class Foo {
public:
    Foo() {cached_ptr = new int;}
    ~Foo() {delete cached_ptr;}

    // error: assignment to member Foo::cached_val of const struct Foo
    void f() const {cached_val = 1;}

    // warning: const cast away: const struct Foo *-> Foo *
    void g() const {((Foo*)this)->cached_val = 2;}

    // ok
    void h() const {*cached_ptr = 3;}
private:
    int *cached_ptr;
    int cached_val;
};

```

The call to `Foo::f` is an error. The call to `Foo::g` is only a warning, but beware, if the compiler placed the `const Foo` in special read only memory, the results of assigning to `cached_val` are undefined. The safest solution is `Foo::h`, but at the expense of one level of indirection.

**Inlining.** Inline functions have function call semantics, and thus are superior to macros for expanding code in line. Used primarily for efficiency by removing function call overhead from small functions, inlines may also be used to allow the C compiler and/or assembler to optimize the expanded code in context. Be aware that `inline` is only a hint to the compiler, and in certain cases where inline expansion is not possible, the function may be outlined as a static function in each .c file in which it is defined. In these situations, the

benefits of inlining can be totally lost without the user's knowledge. Debugging inline functions can also be difficult so some functions are best inlined late in the development cycle when this is no longer a problem. Excellent discussions of inline functions can be found in [5] [6].

**Enumerators in Class Scope.** Using enums defined in class scope logically identifies the enumerators with a particular class and helps keep the global name space clean. If the enum is for internal use only, enumerators declared in the protected or private sections of a class respect access rules.

```
class Foo {
public:
    enum FooColor {red, green, blue};
};
class Bar {
public:
    enum BarColor {blue, red, green};
private:
    enum State {good, bad, indifferent};
};
main()
{
    int i = Foo::red;    // i = 0;
    int j = Bar::red;    // j = 1;
    int k = Bar::good;   // error:
                        // main() cannot access Bar::good: private member
}
```

Note though, the enumeration's type name is in global scope.

```
class Widget {
public:
    enum Color {red, green, blue};
};
class Gizmo {
public:
    // error: Color redefined, enum tag not local to class
    enum Color {blue, red, green};
};
```

**Purely Static Classes.** Classes with all static members effectively provide global data and functions within a completely object-oriented design.

```
class SomeSharedResource {
public:
    static int value() {return i;}
    static void value(int k) {i = k;}
private:
    static int i;
};
```

This class encapsulates the data and operations managing SomeSharedResource. Since all its users are sharing a class rather than an object (i.e. an instance of the class), there is no need to create a globally named object. In fact, the class has no constructors. Initialization of the static data are performed at program startup, and is defined using the syntax:



```
int SomeSharedResource::i = 0;
```

in one and only one source file.

**Assignment Operators.** Assignment operators that perform some destructive act must guard against assigning an object to itself. For instance:

```
class Foo {
public:
    Foo(int i) : sz(i) {anArray = new int[sz];}
    ~Foo() {delete[sz] anArray;}

    Foo& operator=(const Foo& fr);
private:
    int sz;
    int *anArray;
};
Foo&
Foo::operator=(const Foo& fr)
{
    if(this != &fr){          // guard against assignment to self
        delete[sz] anArray;  // disastrous if assigning to self

        anArray = new int[fr.sz];
        for(int i = 0; i < sz; ++i)
            anArray[i] = fr.anArray[i];
    }
    return *this;
}
```

**Coding Standards.** We found a class declaration template, naming conventions and detailed comments in the class declaration very helpful. Since during development, a header file *is* the manual page for a class, it was reassuring to know how it would be organized, and that it would be self explanatory. While the particular template and standards any group of developers might use is probably a religious issue, suffice it to say, it helped us and we would strongly urge their use. Some coding conventions can be found in<sup>[7]</sup>.

### 3. Object-Oriented Design

#### 3.1 Consider your User

We feel that different design criteria may be applied to classes developed for a class library and those developed to implement an application.

Classes distributed as part of a general library must rigorously respect object-oriented principles, especially data hiding. Whether member data and functions are placed in the public, protected or private section of a class is a serious design decision. The class must be "bullet proof" with respect to anything the user may do, and be extensible in ways the original author hadn't imagined.

On the other hand, if the class is part of an application, taking occasional liberties with OO principles is unavoidable. Regardless of the amount of up front planning, resign yourself to the fact that many classes will require re-implementations as project development continues. If you find yourselves spending too much time debating OO philosophy at the expense of your scheduled commitments, settle on something and move on. Since you are the user of your own classes, don't become fanatical in trying to "protect yourself from yourself." Document your compromises, deliver your application to your user, and improve your classes in the next release.

## 3.2 Rules of Thumb

We don't have a rigorous design methodology, but over time developed a few maxims that helped guide high level decisions.

**When in doubt, make it an object.** When beginning to program in C++, we felt something had to be really important before elevating it to the status of class. We were essentially writing "old" C programs with only the major data structures represented as classes. Soon we discovered that smaller groups of independent functions and data could be collected into useful classes. We even developed small classes that did the work of a single function, e.g. check the validity of a string. In the end, we had implemented completely object-oriented designs; neither application had any global data or any global functions (except main).

**Keep it simple.** A class with one area of responsibility, implemented by a collection member functions that each perform a discrete task, is easy to reuse or extend through inheritance. If you find a class doing many disparate things, break it apart into distinctive classes and re-create the desired behavior with these smaller classes.

**Don't make friends too easily.** Friend functions are indispensable for operations such as I/O and binary operators,<sup>[8]</sup> but other uses may suggest a weakness in your design. Is it really necessary to provide the friend access to the class' private members or is the class simply lacking the proper interface and the friend mechanism is a quick short cut?

**Don't ask personal questions.** The temptation to give a some objects a *type field* is almost unbearable. Resist! The virtual function mechanism provides the means for using on an object without "asking" exactly what it is. In general, switch statements based on object type are only needed when the application has lost an object's identity, e.g. when reading a stream of different objects from disk.

## 4. Sales Tax or Inheritance Tax, Either Way You Pay

Higher order classes are constructed by owning its constituent pieces (buying), deriving from its constituent pieces (inheriting), or some combination of the two. While we may someday resolve this decision to buy or inherit using sound software engineering principles, the current state of the art relies on intuition and religious zeal. Based on our experiences, we will offer some guidelines for choosing between buying and inheriting. For this discussion, we call a class that contains only built-in types a *simple class*, and one that contains user-defined types a *complex class*.

A major concern when designing a class, whether simple or complex, is designing its interfaces. All classes must provide a public interface to its functionality, but designers of a complex class must also contend with the interaction between the constituent classes. It is the nature of these interfaces that often drives the buy vs. inherit decision.

### 4.1 The Choice

To construct a complex class by buying its pieces, one important criteria must be satisfied: the complex class must be able to use its constituents as purchased, accessing their functionality via their public interface<sup>2</sup>. In exchange for this isolation from a constituent's internals, any of its facilities the complex class wishes to make available must be explicitly provided in the complex class' public interface.

Inheritance reverses these issues. The complex class is not isolated from its constituents internals as it can access its protected members. Of course this comes at the cost of vulnerability to changes in the base class' implementation. In exchange, the constituents' facilities may be directly accessible to users of the complex class with little or no work by the programmer. In the simplest cast, a complex class may be used directly as an instance of its public base classes. In a nut shell, inheritance provides greater access at the expense of a greater commitment.

2. Of course, this is not strictly true since the friend mechanism can be used to circumvent this restriction.



In our experience, we found three common reasons to build a complex class: *specialization*, *extension* and *aggregation*. In the next sections, we discuss each case, and how it may lend itself to either a buy or inherit solution.

**4.1.1 Specialization.** A class exists and you wish to specialize its use. A classic case involves deriving from an abstract class, i.e. a class that specifies its interface, but does not provide a full implementation. An abstract class may not be instantiated, so by definition, you must inherit. For example, our graphics library specialized the abstract base class `Geometry` to implement specific constructs such as `Line` and `Arc`.

Another type of specialization takes an existing class and limits its functionality. For example, a stack may be implemented as a specialization of a linked list. The main consideration is preventing the user of `Stack` from accessing any of `List`'s functions that do not access its elements first in last out. Whether class `Stack` owns its `List` or is privately derived from `List` is a matter of taste. We chose the latter and defined inline push and pop functions.

**4.1.2 Aggregation.** A class pulls together the facilities of many other classes under one roof. The complex class may aggregate either data or functionality.

**Data Aggregates.** In general, we aggregated data through ownership. Our rather arbitrary and subjective rationale was as follows: since class `Car` cannot inherit from class `Tire` four times, using inheritance to aggregate data is not a general solution and should be avoided. Designing the public interface to a data aggregate poses an interesting question. Should the aggregate provide access functions that return (perhaps const) references to its member data, or should it replicate the member's interface?

Our data aggregates didn't usually have unique behavior in their own right, but simply collected related pieces of information that logically belonged together. If users of an aggregate are free to call *any* (const) function on a constituent, why not give away a (const) reference to it? Or for that matter, why not make it public? We couldn't bring ourselves to do that, but we did give away references to selected member data. As we said in Section 3.1, different design criteria may be applied in different situations.

**Functionality Aggregates.** If the complex class collects its functionality from many sources, inheritance may provide an elegant solution. This use of inheritance is recommended by Meyer <sup>[9]</sup>, and in one example he creates a class `Window` by inheriting from class `Screen_object`, class `Text` and class `Tree`. We have no experience with this technique, but would like to explore the possibilities.

**4.1.5 Extension.** The complex class extends the facilities of an existing class by adding more data and/or more functionality.

**Extending Data.** Extending a class' data is similar to data aggregation, but rather than collecting separate sources of data to be managed together, you wish simply to add data fields to an existing class. This notion may be viewed as a 1-to-1 isA relationship, and is easily implemented with inheritance. For example given:

```
class PartClassification {
public:
    PartClassification(int id) : unique_id(id) {}
    int uniqueId() const {return unique_id;}
private:
    int unique_id;
};
```

the more general `PartDescription` is easily implemented as:

```

class PartDescription : public PartClassification {
public:
    PartDescription(int id, const String& nm)
        : PartClassification(id), my_name(nm) {}
    String name() const {return my_name;}
private:
    String my_name;
};

```

Users of `PartDescription` have direct access to `UniqueId()` as well as `Name()`. Since this extension in no way alters the notion of `PartClassification`, public inheritance poses no problems.

**Extending Functionality.** Extending functionality may also be done using inheritance. Given a library manager class that handles checking data in and out of a library, an interactive library class may be publicly derived, inheriting all the library manager functionality and adding new mouse/icon based user-interface functions. Since the interactive library did not modify the behavior of its base class, public inheritance provides an excellent solution.

There are cases however where the extensions make the new class incompatible with the original, and private inheritance must be used to prevent the derived class from ever being treated as an instance of its base class.

Our Geometry class hierarchy includes class `GeometryGroup`. Publicly derived from `Geometry`, a `GeometryGroup` can be manipulated as `Geometry` (draw, erase etc.), but it needs additional facilities for grouping together a collection of `Geometry*`'s. The declaration:

```

class GeometryGroup : public Geometry, private List_of_p(Geometry)
{
    // ...
    void insert(Geometry* gp)
    {insertion_update(gp); List_of_p(Geometry)::insert(gp);}
};

```

gives `GeometryGroup` its collection of `Geometry*`'s by privately deriving from `List_of_p(Geometry)`<sup>3</sup>. Since the group's `insert` function must do some preprocessing before adding a new `Geometry*` to the list, private inheritance insures `List_of_p(Geometry)`'s non-virtual `insert` can never be called directly. If all of `List_of_p(Geometry)`'s `insert/remove` functions had been virtual, we could have derived publicly and reimplemented them in `GeometryGroup`, but that would have required writing many more functions than we really needed.

## 4.2 The Interface

When creating a complex class it is necessary to consider both the public interface to a complex class' facilities and the interface among its parts.

Design of the public interface poses interesting problems. For public derivation, the interface is easy: the base class' interface is directly available. For private derivation, you have two choices: provide public access to selected base class functions or provide intermediary functions in the derived class that in turn call base class functions. For example:

3. `List_of_p(T)` is a macro invocation generating a type name for a linked list of `T*`'s.

```

class Base {
public:
    void f();
    void g();
    void h();
};

class Derived : private Base {
public:
    Base::f;           // adjust access
    void g() {Base::g();} // provide intermediary
};

```

provides users with direct access to `Base::f`<sup>4</sup> and indirect access to `Base::g`. `Base::h` is hidden from users of class `Derived`.

If the class is built using the "buy" approach, the interface to the aggregate class may be designed in two ways. 1) Provide a member function that returns a reference to the member data. This is quick and convenient (and one we sometimes employed), but violates OO principles<sup>[1][2]</sup>. Or 2) provide the services needed in the aggregate's interface. This requires more work, but is a more robust solution. Consider:

```

class Foo {
public:
    int value() const {return val;}
private:
    int val;
};

class Aggregate {
public:
    Foo& myFoo() {return f;}
    void value() {f.value();}
private:
    Foo f;
};

main()
{
    Aggregate a;
    int j = a.myFoo().value(); // trouble if Foo changes
    int k = a.value();         // safe
}

```

Either solution works, but with trade-offs. Giving away references to member data makes users of class `Aggregate` dependent on its implementation. On the other hand, what if users of `Aggregate` need access to 27 member functions of class `Foo`? Replicating the interface would certainly be tedious, and it's not clear that given so much dependency on `Foo`, a change in `Aggregate`'s implementation is feasible. We used both methods in various places, and until the applications live a longer life cycle, we cannot say what impact on enhancements or maintenance either solution will have.

Intra-constituent communication poses similar problems. Whether a class may be reused as a base class is directly dependent on the class' public and protected interface. As development progressed, we found new

4. Note, in cfront 2.00, this is not implemented for overloaded functions.

uses for classes that had originally been developed to solve a particular problem. Since these classes were not designed with reuse in mind, in the interest of time, we often simply changed `private` to `protected` in the base class' header file and carried on giving the derived class direct access to any facilities it needed. Another possible solution is to redesign the existing base class to provide a protected interface where all access to member data, even within the class' own member functions, is made through function calls<sup>[10]</sup>. This is perhaps the most object-oriented solution, but in practice we did not find it necessary.

## 5. Functionoids and Glue Objects

*Functionoids* and *Glue* are types of objects we used to keep major pieces of the applications independent and reusable.

### 5.1 Functionoids

A *functionoid* is an object that behaves like a function. In the beginning, our objects were data oriented, or nouns. Our prototypes were designed as a collection of *important things* acted on by independent (non-member) functions. In time we realized these independent functions suggested a need for a type of action object, or verb. A *functionoid* is such a verb.

Our user-interface displayed buttons on screen the user "pressed" with the mouse. A common "C" solution is to give the button a pointer to a function the button calls when pressed. Unfortunately, a function requires global data to understand the current state of the system. Our experience is, the more global data an application depends on, the less reusable its code.

As an alternative, our Buttons were given a pointer to the abstract base class `Dispatchable`:

```
class Dispatchable {
public:
    virtual int operator() = 0;
};
```

On a button press, the button calls its `Dispatchable`'s function call operator.

```
class Button {
public:
    Button(const String& label, Dispatchable* dp)
        : l(label) my_dp(dp) {}
    int press() {return (*my_dp)();}
private:
    String l;
    Dispatchable* my_dp;
};
```

The programmer derives from class `Dispatchable` a class to do the appropriate work, creates one, and passes it to the `Button`'s constructor. This derived class may have any necessary member data and functions so it may be kept informed of the system's state. We also experimented with `Dispatchables` that when "executed", stored some state information before changing things, and then were placed on a stack. An undo `Button` popped the stack and called the top `Dispatchable`'s restore function that returned its piece of the world to its stored state. In this way we created an effective undo facility.

Functionoids are useful in situations where function pointers are traditionally used such as performing state transitions in a finite state machine, or as arguments to error handlers. Any tool that eliminates dependence on global information helps making software reusable.



## 5.2 Glue

*Glue* is an object that establishes communication between two objects that must interact but do not expressly know about the others existence. Both applications we developed were CAD tools, each represented as a class, that needed to access data managed by a library manager class that was essentially a tool itself. The philosophy we followed was that the CAD tools didn't need to know what was managing the data and the library manager didn't need to know what object was accessing its data. This was done by creating a glue object that knows about all the objects required to establish this communication.

`LibraryManager` is a large aggregate class that handles data I/O for an anonymous user. Its use is customized by providing `Readers` (a type of functionoid). A `NULL Reader` is valid and indicates do nothing.

```
class LibraryManager {
public:
    LibraryManager() : myReader(NULL) { /* ... */ }
    void setReader(Reader*); // Change myReader
    // ...
private:
    Reader* myReader; // Called when LibraryManager is
                     // asked to read.
    // ...
};
```

`CadTool` is also a large aggregate class that produces data but doesn't really know where the data is stored. Its use is customized by providing `Savers` (also a functionoid.)

```
class CadTool {
public:
    CadTool() : mySaver(NULL) { /* ... */ }
    void setSaver(Saver*); // Set mySaver
    // ...
private:
    Saver* mySaver; // Called when saving
                  // data created in CadTool.
    // ...
};
```

`CadLibGlue` does most of its work in the constructor, creating the proper functionoids and giving them to the `LibraryManager` and `CadTool`.

```
class CadLibGlue {
public:
    CadLibGlue(CadTool& c, LibraryManager& l);
                                // Creates and sets the functionoids
                                // needed to customize the interface between the
                                // given LibraryManager and CadTool.
    ~CadLibGlue(); // Deletes all objects created in constructor.
    // ...
private:
    CadToolA& cr; // the CadTool I am managing
    LibraryManager& lmr; // the LibraryManager I am managing
    Reader* rp; // a Reader for the LibraryManager
```

```

    Saver* sp;                // a Saver for the CadTool
    // ...
};

```

Glue classes make program development easier since each of the large classes that it connects are not dependent on changes to the other classes. This independence also makes the large classes more reusable.

## 6. Conclusions

Looking back, we would say Bjarne was right, the learning curve is long, but working in C++ we developed tools faster and of a higher caliber.

Climbing the C++ learning curve is a challenging undertaking. While there is new language syntax to learn, it is developing an "object-oriented" way of thinking that requires considerable effort and practice. Once the notion of objects starts to click (remember anything can be an object) the advantages of C++ become more and more evident. We feel the investment in learning C++ was worthwhile.

## 7. Acknowledgements

We would like to thank Rob Murray, Nancy Wilkinson, Terry Weitzen and all the others for a great class library we used to jump-start our project. Having a code base to build on was a tremendous help. Thanks to Jonathan Shopiro for a willing ear and sagely advice, to Judy Ward, who made developing software with a beta compiler a workable (if not exciting) proposition and to Martin Carroll for a nice set of development tools. Oh, and thank you Bjarne for encouraging us to take on this project in C++ in the first place.

## REFERENCES

1. *Object-Oriented Programming: An Objective Sense of Style*, K. Lieberherr, I. Holland, A. Riel, OOPSLA '88 Proceedings.
2. *Comments on "the Law of Demeter" and C++*, Markku Sakkinen, SIGPLAN Notices, Vol. 23, No. 12.
3. *References In C++*, Martin Carroll, C++ Report, Vol. 1, No. 1, January, 1989.
4. *References in C++*, Andrew Koenig, Journal of Object Oriented Programming, March/April 1989, Vol. 1 No. 6.
5. *Inline Functions in C++*, Dennis Mancl, C++ Report, Vol. 2 No. 2, February, 1990.
6. *Using Inline Functions Effectively, Parts 1 and 2*, Mark Rafter, C++ Report, April 1989, Vol. 1 No. 4 and May 1989, Vol. 1, No. 5.
7. *A Style for Writing C++ Classes*, Peter A. Kirsliis, USENIX 1897 C++ Workshop Proceedings.
8. *What Are Friends For*, Andrew Koenig, Journal of Object Oriented Programming, Nov./Dec. 1989, Vol. 2, No. 4.
9. *Object-Oriented Software Construction*, Bertrand Meyer, Prentice Hall, 1988.
10. *Variables Limit Reusability*, Allen Wirfs-Brock, Brian Wilkerson, Journal of Object Oriented Programming, May/June 1989, Vol. 2 No. 1.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

## Editorial

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

## Articles

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

## Index

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.

The first issue of the journal was published in January 1991. It contained several articles on the state of the art of C++ and its future development.



# Designing C++ Libraries

James M. Coggins  
Computer Science Department  
University of North Carolina at Chapel Hill  
coggins@cs.unc.edu  
and

Center of Excellence in Space Data and Information Science  
NASA Goddard Space Flight Center, Code 630.5

## Abstract

Class libraries encapsulate useful implementation ideas. Designing libraries is difficult because of competing, conflicting objectives. Design criteria are needed to guide library architects toward good encapsulations. This paper argues that three strategies used in many object-oriented libraries are actually poor library design criteria, particularly in C++. A new criterion is defined and examples of its use are illustrated. The new criterion applies the common software engineering design principle "separation of concerns" in a specific way that leads to effective library designs. The approach leads to a model of collaborative development of class libraries customized for a family of applications.

## The Challenge of Software Library Design

Design is the art of recognizing, evaluating, and selecting tradeoffs. In software development, design involves selecting from the multitude of expressions available those particular forms that result in "good" code - code whose structure, function, and operation optimize certain criteria. Software design is difficult: the designer must weigh arbitrary, interacting constraints and conflicting objectives. This difficulty is essential to the enterprise, not an incidental, temporary lack of technological development, but the incidental difficulties are not negligible [1]. Tools, languages, or techniques that ease the incidental difficulties of expressing, understanding, and updating ideas embodied in software are badly needed. Object-oriented design provides such relief by specifying some criteria concerning methods of encapsulation and a semantics of inheritance that have been incorporated into several new programming languages, including C++ [10, 16].

To say that a library is written using object-oriented design says something about the mechanisms used to code and package the library in a programming language but nothing about the criteria used to select and evaluate the tradeoffs that form the library. Library writers and their clients must understand that an "object-oriented library" does not mean a "well-designed library"; object-oriented design is not a sufficiently comprehensive or detailed discipline to support such a claim [14, 17].

Moreover, C++ leaves the programmer the maximum flexibility of expression - a language design criterion inherited from C [16]. This is good in that there are many alternative strategies by which a programmer can express an elegant idea; in fact there are so many options provided in C++ that there is probably an elegant expression available for almost any worthwhile idea. The flexibility of C++ is bad in that programmers are too often tempted to select poor tradeoffs because they do not understand the multitude of options that are available.

Regrettably, the ignorance on the part of computer programmers concerning good design is compounded by the lack of a culture in which programmers study good designs and communicate design ideas to others for review [19]. We tend to want to learn from experience rather than from

study. We are ready to adopt anything that works, especially if it can be obtained free. The growing array of options available through object-oriented programming and personal workstations with color bit-mapped displays (a fertile application field for object-oriented design) overwhelms the ability of psychology to provide scientific data on how this new programming and user interface technology might be used effectively, reliably, and safely. Unfortunately, experience is an effective but inefficient teacher. "Newton said he saw farther because he stood on the shoulders of giants. Computer programmers stand on each others' toes."

The result is that there is some truly ridiculous code being written out there; all of it, however, written in one of the popular new languages, and all of it adhering to (some form of) the Object-Oriented Programming Discipline (as if this Discipline were carved in the side of a mountain - in fact, OOP is not a discipline at all, but a convention [7,14]). I have been able to trace every ridiculous implementation question I have been asked to the ridiculous design concept that inspired it. If we were better designers, we would know that *encountering the need for an outrageous hack during implementation is an indictment of our design concepts, not a challenge to our hacking skills.*

Because library designers have so many options and so little experience with them both individually and collectively (even if we were in the habit of using our collective knowledge effectively), we need to specify, evaluate, and select specific criteria that will govern how we evaluate the tradeoffs we face in the design of object-oriented libraries. Informed consumers of such libraries should demand of the library supplier some discussion not only of what is in the library and how it works but also of what criteria were used to select what is in the library and why it is expressed as it is. Users should seek libraries that provide appropriate abstractions for their application areas, not simply code that can be reused.

C++ is a new programming language advocating new software design methods and programming language technologies, so it is not surprising that there has been considerable discussion concerning how the language and the expressions it supports can be employed most effectively. Several C++ libraries embodying different design objectives and philosophies have become well-known in the community, but none have received uniformly positive reviews. Further discussion of how to develop useful, effective libraries in C++ is still required.

This paper will review three design strategies used, even advocated, for C++ libraries, discuss serious problems with all three approaches, and propose an alternative set of design criteria, more specific than the three common approaches, with examples from the author's C++ library. The three design strategies current in the practice and literature of object-oriented software design are

- (1) comprehensive, monolithic class hierarchies,
- (2) collections of tiny encapsulated tools, and
- (3) flow control residing in the user interface code.

The following sections critique these strategies.

### **Why comprehensive, monolithic hierarchies are a bad idea**

The notion of defining a single hierarchy containing all of the classes one might ever need was implemented in the Smalltalk system [8]. Every class is a part of a single inheritance hierarchy with object as its root. The Smalltalk design has been transplanted into other object-oriented languages such as Objective-C's ICpack 201 [15] and even C++. The NIH class library (formerly called OOPS) is a transliteration of a large subset of the Smalltalk hierarchy into C++ [9]. The ET++ library also consists of a monolithic class hierarchy.

Smalltalk is an interpreted language and environment that gives up a great deal of performance in order to achieve an amazing degree of flexibility. Whether this degree of flexibility is really

good or not is beyond the scope of this paper, and does not have a straightforward answer. Since the definition of the Smalltalk system is accessible to the programmer, each programmer can create customized but incompatible Smalltalk environments. The main reasons for using object-oriented languages other than Smalltalk is that performance, portability, and compatibility matter. The tradeoff one accepts in selecting a more conventional language is that changing or customizing one's environment is more difficult than in Smalltalk. A monolithic class hierarchy is less troublesome in Smalltalk than in other languages because the Smalltalk environment can be changed so easily. More conventional operating system and language environments are much more difficult to change. Thus, *imposing a monolithic hierarchy locks users into a single implementation of each class that is difficult to change.*

The monolithic hierarchy is particularly inappropriate in C++. Implementation of a monolithic hierarchy in C++ requires extensive use of macros to fake polymorphism and complicated class names to encode type information in order to satisfy the strong type checking of C++ [9]. The result is object-oriented code that is more error-prone, harder to read and understand, with even more memory burden on the programmer than with conventional C libraries due to even more complicated names to remember. Rather than helping programmers to control programs, monolithic hierarchies combine with design decisions implemented in C++ in unexpectedly nasty ways to produce more of a mess for programmers than we faced with C. *C++ is not Smalltalk.*

Monolithic hierarchies impose considerable overhead in terms of memory load, compilation and link time, run time, code size, and hidden dependencies. This contradicts the C++ design principle *"You shouldn't have to pay for what you don't use."*

Moreover, the use of a monolithic class hierarchy is contrary to the design principles of C++. C++ was designed to provide as many implementation options as possible within its constraints, not to dictate solutions or solution methods [16,17]. To use C++ in a way that restricts those options in costly, ineffective ways is ironic. The experience of many C++ users doing real work is that monolithic hierarchies provide a false economy. *Code reuse is no bargain if the price to be paid is added source code complexity and heavier memory load for library users.*

The costs of using monolithic hierarchies in overhead, inflexibility, conflicting objectives, and code complexity makes monolithic hierarchies an inappropriate design strategy for C++ libraries.

### **Why a toolkit of tiny classes is a bad idea**

The worst aspect of object-oriented design is that it tends to encourage bottom-up design instead of top-down design [17]. Top-down design (also called Stepwise Refinement [20]) is a heuristic that suggests that better software systems result if the application-level concepts guide the design process rather than the low-level implementation structures that happen to be available or convenient [2,7,20]. Another way to say this is that software development should be driven by needs, not by capabilities.

It has been claimed that real systems have no "top". The Tower of Babel had no top. The Strategic Defense Initiative has no top. Real, implementable systems do have "tops" that are specified and guided by a single mind, or a few of them. This architectural vision gives the product conceptual integrity [2]. While a bottom-up *realization* might make module testing easier in practice, this does not diminish the necessity of an *architectural* vision to guide decisions at lower levels of abstraction concerning what to implement and what to ignore. The tendency of object-oriented methods to favor the bottom-up approach causes the greatest disasters when it leads programmers to design libraries of tiny classes from which, supposedly, larger classes may be constructed. This approach to code reuse is bound to fail. The fallacy of this approach is easy to see from several viewpoints.



From an economic viewpoint, to buy a class (that is, to obtain, learn about, and use somebody else's class definition and code) saves the buyer certain costs in developing and debugging the code while exacting certain other costs: First, the author's approach must be understood and accepted, including its impact on the *buyer's* subsequent design decisions. This cost is substantial, especially when the library developer does not provide coherent design and organization principles for the library. Second, the buyer must relinquish a measure of control over the product; the buyer is now dependent to some extent for bug fixes, enhancements, and compatibility on the author of the library. The economic point of view says that the value of the library must exceed these costs or it will not be bought. From this viewpoint, building collections of tiny classes does not maximize the likelihood that other people will buy the library because *Small encapsulations provide small benefit.*

From a user viewpoint, also, the toolkit of tiny classes is not the most desirable way to obtain labor-saving software. The user has a problem to solve, and the solution requires code to be written. What the user wants is an encapsulation of the concepts required to solve the problem, not a set of, for example, polymorphic stacks and queues that exact heavy costs in memory load and system overhead. The most valuable kind of software from the user's viewpoint is a class that matches the need, not a collection of low-level implementation hacks to which the needs must be retrofitted. I know clients who need encapsulations of radiation beams, anatomical organs, images, the surface of the cornea, the optic disk, galaxies, spectra, land cover classes, random variables, cars on a freeway, and more. I don't know any clients who need polymorphic stacks and queues. *Tiny classes are not what the user needs.*

From a software-engineering viewpoint, a toolkit of tiny classes employs object-oriented design in a way that fails to achieve one of the main reasons for adopting object-oriented design: organizing the name space. The library of tiny classes fails to decrease the surface area of the library (the number of names accessible at some point in the code [6]) and, in the context of strong type checking, complicates the names that are used. *A name space makes a poor database.* This is exactly why conventional libraries with flat name spaces are difficult to use as the library grows larger. A library of tiny classes encounters the same problem, only slightly later because even small encapsulations do contribute a little to the organization of the name space. The library of tiny classes, only delays for a little while the onset of the same problem of mushrooming name spaces that we have experienced for years. *Building a library of tiny classes is a bad design strategy.*

This discussion has important implications for teaching object-oriented design. For one thing, it implies that the common example of inheritance where rectangle and triangle are derived classes from polygon reflects a bad design strategy because the classes are too small to be of any real benefit. Such examples adequately display the mechanisms of encapsulation and inheritance, but to use them as examples of library design is a grave error. Also, the failure of the library of tiny classes implies that object-oriented systems should be built around larger application-area concepts, not around computer programming tricks.

### **Why UI-controlled implementation is probably not a good idea**

Computer people are a friendly, cooperative folk until we enter the arena of user interface design. On this topic, programmers are passionate, interested, parochial, bigoted, and deeply ignorant. And the situation is not getting better. The appeal of user interface design is clear: the immediate, visible results and the unlimited potential for midnight hacking are powerful aphrodisiacs in this community. In addition, a user interface package works with ideas that fit naturally into an object-oriented design methodology, so object-oriented user interface work is aesthetically satisfying.

Color bit-mapped displays have opened new worlds for user interface designers to explore. The complex software systems that control these displays consume many compute cycles and require much attention from the application software; so much attention that many people are advocating that the operation of application programs be controlled by the user interface. That is, after initialization (and the required initialization is extensive), control is given to the user interface system which invokes application procedures via "callback functions" as required by the user interaction [11,18].

This design strategy gives a deserved importance to the user interface and facilitates highly responsive user interaction since nothing happens until the user commands it; the application is interrupt-driven based on the user's actions. I regret that I must report flaws in this approach, but the main flaw is less serious than with the other two design strategies, thus the heading of this section is less negative than that for the previous sections.

There are several tradeoffs in using the UI-controlled design strategy. The user interface control system must be extremely complex in order to handle the variety of interactions that might be required. The structure of the desired user interface must be communicated to the user interface control system, of course, but in addition, the structure of the *application program* must be communicated to the user interface through callback functions. Thus, the initialization of the user interface system is extremely complex, error-prone and tedious.

The problem is that the user-interface control strategy does not map into an object-oriented design. The user interface is not an "object" any more, and it is not separated from other aspects of the application. The structure of the application must be defined to the user interface controller, adding a burden of messy, error-prone procedure calls atop the application. In effect, the application must be written and then its structure must be "explained" via the user interface definition and the callback functions to the user interface control system. While the user interface control system might be written in an object-oriented manner, its appearance to its client applications is like that of kudzu: intertwined, going where it shouldn't, and next to impossible to manage neatly. *UI-controlled implementations are not object-oriented implementations.*

In my work, I have often sacrificed immediate interaction in favor of a simple, object-oriented user interface. Since most of the operations I am interested in are going to take time anyway, I have created a simple, abstract user interface class that appears in programs as an object. For time-critical operations, I allow the user interface class to maintain control and return the results of the whole time-critical user interaction sequence. Encapsulating a simplified user interface this way has allowed rapid development of software tools without the irrelevant distractions of developing a large number of fancy user interface gadgets. By allowing the application area to drive the software development, we have avoided the time sinks of software conversions and of building nice but irrelevant user interface hacks. Our resulting user interface is effective but not fancy. It provides what we need and no more, in a form that is consistent with the rest of our object-oriented research library.

### **Objectives of Library Design**

The purpose of a library is to encapsulate useful implementations in an easy-to-use form. The principal design objective of a library architect is to create and exploit a separation of concerns [12,13] between the library user and the library implementer. The architect endeavors to minimize instances in which it is necessary to know about the internal structure or operation of modules in the library in order to effectively use the library. Object-oriented programming languages contribute to library design by supporting the separation of concerns created by the library architect. However, this does not mean that an object-oriented library is necessarily a well-designed library. Some further guidance is necessary to improve the chances of obtaining a well-designed library.



One form of guidance involves understanding what kinds of encapsulations might be implemented in the library. My library, COOL [4], contains four kinds of encapsulations (Figure 1). **Data Structure encapsulations** are the traditional "objects" in which a class contains a data structure and procedures that operate on it. Member functions might provide the user with a value, allow the user to set a value in the object, or invoke methods in other objects. **Process encapsulations** (I also call these "enzymes") embody processes or modes of interaction between objects of other classes. (I have heard of some people not accepting such structures as valid classes. Remember: algorithm is a noun.) Process encapsulations accept an object and some kind of "go" signal and produce as output a new object. The operation of the process encapsulation might be adjusted by parameters settable using member functions of the process encapsulation or by parameters to constructors. **Device encapsulations** provide an abstract, customized user interface for controlling devices. Such classes are designed to be convenient for users and can provide any desired level of flexibility by using flags and switches controlled by member functions of the class. The complexity of the underlying device (socket, scientific instrument, display, window management library, disk drive, joystick) is hidden, and the device encapsulation provides exactly the capability that is needed in a form that is convenient to use, ignoring unnecessary bells and whistles on the device. **Interface encapsulations** provide interface and conversion procedures between other classes. A class for creating network packets from another class containing a large data structure would be a typical interface encapsulation.

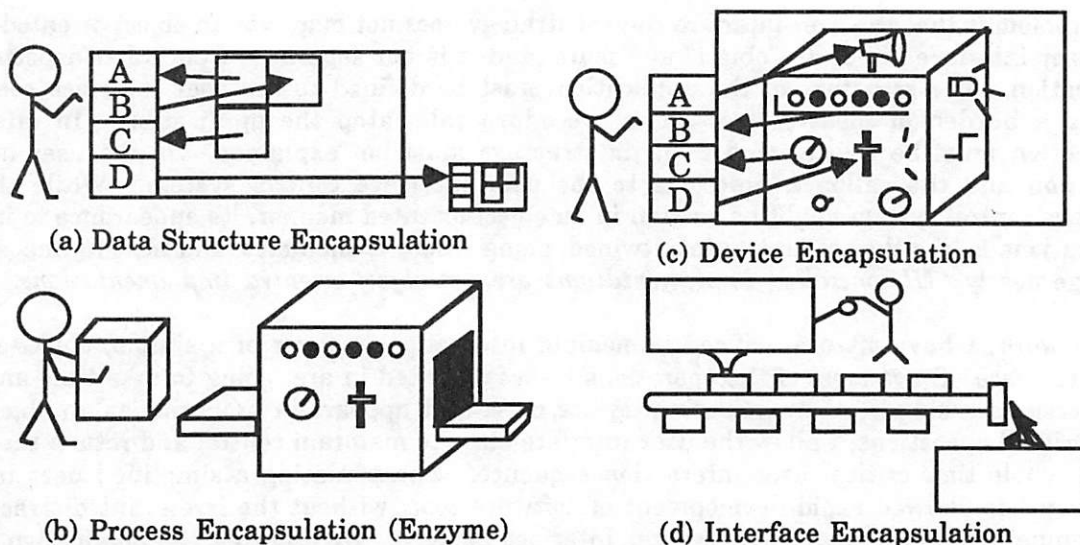


Figure 1: Encapsulation types in COOL

A more important kind of guidance for library developers involves criteria for deciding what kinds of classes should be developed and what kinds of member functions should be provided for classes. I have found that there are three different kinds of concerns that face library developers and users. First are the concerns of **processing**, including (1) how objects might be teased apart in the computer's memory to form a complex internal structure that facilitates processing and (2) what process encapsulations are required to get the job done. Second are the concerns of **display and interaction** where one encounters many hardware dependencies and where timing of interrupts and input sampling is likely to be critical. Third are **storage and communication** concerns where formatting of bytes in a standard sequential order is an essential problem.

I submit that designing class libraries to separate these concerns leads to elegant, effective, and useful encapsulations.

## Library Design in Practice

This new design criterion immediately affects how one selects and implements encapsulations in a library. Examples from my library, COOL [4,5], which is concerned with image processing, image pattern recognition, and computer graphics, illustrate this design criterion in practice.

The image class is central to my library. I originally wrote the image class with `load()` and `save()` member functions so that an image knew how to load itself from disk and save itself there. These were the longest member functions in the whole image class, so they already looked out of place there. When I later realized that I needed to `load()` and `save()` certain matrices, I found that the code I had developed to implement loads and saves of images could not be used (via inheritance, for example) to help me define analogous operations on matrices. This struck me as a serious design flaw. I decided to bring together all disk handling routines under a separate inheritance structure in which basic disk file services are provided by a base class, `diskfile`, which conveniently encapsulates UNIX file handling procedures, while file formatting information for different kinds of files is hidden in subclasses of class `diskfile`. Class `textfile` is a subclass of `diskfile` that has additional operations and subclasses of its own. This design permits real code sharing among `diskfile` and its subclasses, and the separation of concerns permits users to understand what is going on. This small inheritance hierarchy also illustrates an effective use of arguments to base class constructors.

The representation of an image on disk is encapsulated by class `imagefile`. To read an image from disk one creates an `imagefile` with appropriate parameters and then invokes

```
image imagefile::load(int plane=0).
```

The image returned will be a plane from the `imagefile` (which can hold many image planes). Similarly, to save an image to disk, one invokes

```
void imagefile::save(image im, int plane=0).
```

The `imagefile` object does not need to know the internal structure of an image - it does need to know how to ask an image for whatever components of the image are required to create a disk representation of it. Since the internal representation of an image is essentially the same as its disk representation (a raster scan array of pixels), the effect of this approach in this instance is not dramatic. Suppose however, that we needed to manipulate sparse matrices as persistent objects. For efficient operation, a sparse matrix needs to be represented in a multilinked list form. For storage, however, a compact, sequential form is required. Under my library design criterion, the processing class `sparse_matrix` is separate from the storage class `sparse_matrix_file`. To store a `sparse_matrix` to disk, one creates a `sparse_matrix_file` and sends it the message

```
void sparse_matrix_file::save(sparse_matrix& sm).
```

The `save` method could either ask the `sparse_matrix` for a compacted representation or create one by asking the `sparse_matrix` for its elements in turn using the same access routines other clients of the `sparse_matrix` class use. I prefer the latter approach since the serialization of the data structure, which is irrelevant to processing but is an essential communication and storage concern, is controlled by the storage class. This approach does not require the storage class to understand the details of the processing class' structure: it simply requires that a "next item" be defined and that relevant information about that item be accessible somehow, preferably through member function calls. This capability is probably already present for use by clients; anyway.

These examples demonstrate that the separation of concerns between processing classes and storage classes affects the selection of what classes need to be implemented and provides a consistent method for providing the storage services for new classes. The next examples show how the separation of concerns guides the design of display and interaction classes and how abstracting display facilities makes conversion to new platforms easier.

Another example of the separation of concerns criterion affecting library design occurs in my `imagetool` and `Ximagetool` classes, which encapsulate image display on workstations under SunView [18] and the X Window System [11], respectively. These classes are device encapsulations that hide massive user interface libraries and present just the customized capabilities I need for image and graphics displays. To display an image on a workstation running X, one creates an `Ximagetool` object and invokes the member function

```
void Ximagetool::display(int viewport, image im).
```

The `viewport` argument specifies the quadrant of the `Ximagetool` window in which the image will be displayed. The contribution of the `imagetool` and `Ximagetool` classes is to abstract the essential qualities I need in an image display, on whatever device the display appears. All three of my image display classes (the third is called `ikonas`) have identical public interfaces, but their internal structures and the systems they control are very different. My library design criterion requires that display and interaction be separated from processing concerns, so the display of images is encapsulated and separated from the code that is concerned with the processing of images. By encapsulating the code that is dependent on the display system, I know exactly what needs to be rewritten to port my image display to another device or another user interface control system. The port of `imagetool` to X took just a few hours, and there was never any doubt that everything that needed to be ported was indeed ported.

The separation of concerns permits definition of classes whose purpose is to control transformations of objects or interactions between objects. Such process encapsulations or enzymes encapsulate algorithms or protocols rather than data structures. An excellent example of this kind of class is my `fft_server`. The Fast Discrete Fourier Transform is an intricate algorithm that is a worthy subject for study itself. My `fft_server` class encapsulates this algorithm. Objects of the class are created for each particular size of FDFT that is required. The constructor precomputes tables to make bit-reverse shuffles and butterflies run faster when the processing takes place. These tables, whose size depends on the size of the object being transformed, include sine and cosine computations for the complex exponentials required for the appropriate size of FDFT algorithm.

The following example illustrates how storage, processing, and display classes interact in COOL. The code segment reads an image and a filter from disk, transforms the image, applies the filter, and displays the image before and after filtering. All appropriate type conversions are performed automatically when they are needed.

```
Ximagetool screen;                // create the display object

imagefile imfile("input.im");     // open the image file for reading
image im = imfile.load();          // load the image into memory
screen.display(0,im);              // display the image on the screen

fft_server FFT(im.shape());        // define an fft_server with correct
                                   // size for the input image

image spectrum = FFT.forward(im);  // transform the image

imagefile filterfile("filter.im"); // open the filter file
image filter=filterfile.load();    // load the filter into memory

spectrum.mpy(filter);              // apply the filter to the spectrum

image inverted = FFT.inverse(spectrum); // inverse transform

screen.display(1,inverted);        // display the result of filtering
```



When a procedure should be implemented as an enzyme class rather than as a member function? If the procedure is general enough to interact with several existing classes, it should be an enzyme. If the procedure is compute-intensive and could be optimized for speed by precomputing data in the constructor, then it should be an enzyme. If the procedure is experimental and therefore is subject to frequent or drastic change, then it should be implemented as an enzyme in order to isolate the changes from the rest of the library. If the operation of the procedure depends on the settings of a group of parameters, it might be implemented as an enzyme to keep the client class free of the clutter of the parameters and the member functions to manipulate them.

A kind of class that will soon be part of COOL is a device encapsulation that implements remote servers for processing compute-intensive image operations. One class encapsulates network operations, including establishing a connection to the remote server and sending packets to the server. An object of this class represents the client end of the network. An interface class defines the packet types for data to be sent to the server. Providing the interface object an image and a pointer to the network object results in appropriate packets being transmitted to the remote server. Similarly, results from the server appear as packets in the network object that are interpreted through the interface object to yield an image.

Now we consider a possible future addition to COOL that appears at first to raise problems for my library design criterion. Suppose we want a class, `net_display`, that can display an image either on the local workstation or on a remote workstation in a manner that is transparent to the user. This class must involve functions in both the *display and interaction* category and in the *storage and communication* category. Does this violate the separation of concerns? No, the desired encapsulation is a reasonable one: a design criterion that forbids useful encapsulations should be sharply questioned. However, a poorly designed implementation of this class might seek to violate the separation of concerns or even perform outrageous hacks of the C++ runtime system. A good implementation will maintain the separation of concerns as illustrated below.

Suppose we have defined a communications class `C` that handles the transfer of images to a remote device where a server displays the image and a local display class `D` that displays images on the local workstation. The `net_display` class can be designed as follows:

```
class net_display
{
    C* comm;      // note: pointers don't allocate the object
    D* local;     //      they point to.
    int which;    // 0=local display, 1=remote display

public:
    // The first constructor creates a local display;
    // the second creates a remote display
    net_display(<various display params>);
    net_display(char* machine_name, <various display params>);

    // The display function sends the image to comm or local
    // depending on the value of which.
    void display(image im);
}
```

A `net_display` object uses an object of a communication class to perform communication and a display and interaction object to perform display. `net_display` is actually an enzyme class that mediates between the two. This implementation provides the desired function without any hacks, while maintaining the separation of concerns.

## Conclusion: Toward a Collaborative Class Design Methodology

The most important potential impact of Object-Oriented Design is not code sharing, at either the level of statements or the level of broader abstractions. Code sharing can provide only an incremental improvement in the effectiveness and productivity of software engineers. Instead, the most important impact of Object-Oriented Design is to raise the level of abstraction implemented in libraries to a level where our clients can effectively contribute to the design of the software they need. Class design should begin with the client's own description of what the client would like the computer to do. The important concepts in that description are the classes that need to be implemented, whether they are principally nouns (resulting in traditional "classes") or verbs or operators (resulting in process encapsulations) or protocols for using complex devices (resulting in device encapsulations).

Effective library design requires explicit design criteria that aim toward producing larger-grained classes that encapsulate concepts that are meaningful to users. This paper has criticized three common design strategies used in C++ libraries and has presented a new candidate design criterion motivated by a particular separation of concerns.

Effective libraries are most likely to be produced in a collaborative effort between the library architect and an application area specialist. Only by involving application specialists, who are the users of the library, can we identify essential concepts for creating effective encapsulations. Such collaborative efforts keep the library architect focused on issues that yield real progress and ensure that the effort invested in library design will indeed be valuable to a real user. The time has come and the tools are at hand to make this kind of collaborative software design feasible. The separation of concerns among architecture, implementation, and realization provided by object-oriented design, and by the structure of C++ classes in particular, elevate the level of discourse about software design to the application level where our clients can effectively contribute. The role of the library developer resembles that of the medieval swordsmith: our success is measured by the longevity (productivity nowadays) of the client [3].

Well-designed class libraries for families of applications could make possible the software economy [6] that object-oriented design has so far largely failed to create. The lack of guidance for library design has permitted library developers to establish flawed objectives, creating collections of tiny classes that are not really useful or constructing comprehensive monolithic hierarchies that attempt to provide too much generality at the expense of flexibility, control and performance. An intermediate approach is more likely to be effective, especially in the compilation environment of C++, and the intermediate approach that yields the greatest benefits is to create larger encapsulations of concepts that are meaningful to our clients. The resulting libraries, produced by collaborative efforts between clients and computer scientists, are our best hope for producing widely usable tools that affect large numbers of clients in effective, powerful ways.

## Acknowledgements

This work is supported in part by the Center of Excellence in Space Data and Information Science via a contract through the Universities Space Research Association. Several dozen students over the past three years have contributed to the development of COOL by coding, arguing, criticizing, and using the library. Their contributions are deeply appreciated.

## References

1. Brooks, F. P, "No Silver Bullet: Essence and Accidents in Software Engineering," *IEEE Computer*, vol 20, no. 4, April 1987.



2. Brooks, F.P, *The Mythical Man-Month: Essays in Software Engineering*, Addison-Wesley, 1970.
3. Brooks, F. P, "The Computer 'Scientist' as Toolsmith: Studies in Interactive Computer Graphics" In *Information Processing 77*. B. Gilchrist, ed. North Holland:Amsterdam. 371-373.
4. Coggins, J. M, *The COOL Library: Concepts and Facilities*, Tech Rep CO-01, SoftLab Software Systems Laboratory, Computer Science Department, University of North Carolina, 1989.
5. Coggins, J. M, *The COOL Library: User's Manual*, Tech Rep CO-02, SoftLab Software Systems Laboratory, Computer Science Department, University of North Carolina, 1989.
6. Cox, B. J., *Object-Oriented Programming: An Evolutionary Approach*, Addison-Wesley, 1986.
7. Freeman, P. *Software Perspectives: The System is the Message*, Addison-Wesley, 1987.
8. Goldberg, A and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley 1983.
9. Gorlen, K. "An Object-Oriented Class Library for C++ Programmers", *Software: Practice and Experience*, vol. 17, no. 12, December 1987, p. 899.
10. Lippman, S. B., *C++ Primer*, Addison-Wesley, 1989.
11. Nye, A., *Xlib Programming Manual for Version 11 of the X Window System*, O'Reilly and Associates, 1988.
12. Parnas, D. L, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, no. 12, Dec 1972, pp. 1053-1058.
13. Parnas, D. L, "On the design and development of program families," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, March 1976, pp.1-9.
14. Rentsch, Tim, "Object-Oriented Programming", *SIGPLAN Notices*, vol. 17, no. 9, September 1982, pp. 51-57.
15. Stepstone Corporation, *ICpack 201 Tutorial Examples Manual*, Stepstone, 1988.
16. Stroustrup, Bjarne, "The C++ Programming Language", Addison-Wesley, 1986.
17. Stroustrup, Bjarne, "What is Object-Oriented Programming?", *Proc. 1st European Conference on Object-Oriented Programming*, Paris, 1987.
18. Sun Microsystems, *SunView Programmer's Guide*, Sun Microsystems, 1986.
19. Weinberg, G. *The Psychology of Computer Programming*, Van Nostrand Reinhold, 1971.
20. Wirth, N., "Program Development by Stepwise Refinement", *Communications of the ACM*, vol. 14, no. 4, April 1971, p. 221.

1. The first step in the process of creating a new program is to determine the requirements of the program. This involves talking to the user and understanding what they need the program to do. Once the requirements are clear, the next step is to design the program. This involves creating a plan for how the program will be built, including the data structures and algorithms that will be used. The third step is to write the code. This involves translating the design into a form that the computer can understand. The final step is to test the program. This involves running the program and checking to see if it does what it is supposed to do. If there are any problems, they need to be fixed before the program is ready to use.

2. The second step in the process of creating a new program is to design the program. This involves creating a plan for how the program will be built, including the data structures and algorithms that will be used. The third step is to write the code. This involves translating the design into a form that the computer can understand. The final step is to test the program. This involves running the program and checking to see if it does what it is supposed to do. If there are any problems, they need to be fixed before the program is ready to use.

3. The third step in the process of creating a new program is to write the code. This involves translating the design into a form that the computer can understand. The final step is to test the program. This involves running the program and checking to see if it does what it is supposed to do. If there are any problems, they need to be fixed before the program is ready to use.

4. The fourth step in the process of creating a new program is to test the program. This involves running the program and checking to see if it does what it is supposed to do. If there are any problems, they need to be fixed before the program is ready to use.

5. The fifth step in the process of creating a new program is to fix any problems that are found during testing. This involves making changes to the code and re-running the program until it is working correctly.

6. The sixth step in the process of creating a new program is to document the program. This involves writing a manual for the program that explains how to use it and what it does.

7. The seventh step in the process of creating a new program is to release the program. This involves making the program available to the user and letting them know that it is ready to use.

8. The eighth step in the process of creating a new program is to maintain the program. This involves keeping the program up to date and fixing any problems that arise after it has been released.

9. The ninth step in the process of creating a new program is to evaluate the program. This involves checking to see if the program is meeting the user's needs and if it is working well.

10. The tenth step in the process of creating a new program is to improve the program. This involves making changes to the program to make it better and more useful.

# Reliable Distributed Programming in C++: The Arjuna Approach

Graham D Parrington

Computing Laboratory,  
University of Newcastle upon Tyne,  
UK

(Graham.Parrington@uk.ac.newcastle)

## ABSTRACT

Programming in a distributed system is fraught with potential difficulties caused, in part, by the physical distribution of the system itself. By making the distribution of the system *transparent* it is hoped that the task becomes comparable with that of programming a more traditional centralised system. Object-oriented programming systems are natural starting points for such an attempt due to the inherent modularisation and encapsulation properties they possess. *Arjuna* is one such system, programmed in C++, which permits the construction of reliable distributed applications in a relatively transparent manner.

Objects in *Arjuna* can be located anywhere in the distributed system and are accessed as if they were purely local to the application. The use of remote procedure calls to perform the actual accesses is hidden by the use of stub generation techniques which operate on the original C++ class descriptions thus furthering the illusion of transparency. Reliability is achieved through the provision of traditional atomic transaction mechanisms implemented using only standard language features.

## 1. Introduction

Although the physical construction of a distributed system is relatively easy today, the programming of such a system remains a complicated task even for the most accomplished of programmers. In addition to the many additional ways in which distributed systems can fail when compared to their more traditional centralised brethren, the complexity of the communications protocols required to make use of remote resources is often daunting. Consequently many researchers have attempted to overcome these burdens either by providing the illusion that programs are still executing in a centralised environment by making the distribution of the system *transparent* to the programmer, or alternatively by adapting familiar programming techniques and metaphors to the distributed environment (for example, extending the procedure call notion to that of remote procedure call).

Designing and building large complex applications is itself extremely difficult requiring discipline on the part of both designer and programmer. Since such systems are usually too large to be understood in their entirety by a single person, they must be designed and implemented as a set of smaller pieces, each of which is itself sufficiently small to be comprehensible. Many disciplines, some with formal underpinnings, are available to aid this process of

decomposition. One such technique that has gained substantial popularity is the object-oriented programming technique whereby the system is partitioned into a set of logical objects that interact with each other to achieve the required system functionality. Each such logical object is self-contained and provides a well-defined interface that permits the orderly interaction between the object and any other objects in the system. Using this paradigm results in systems that are inherently modular, and since each object is self-contained the object-oriented programming paradigm supports the notions of data abstraction and information hiding directly. It is these properties that make object-oriented systems a natural choice for exploitation in programming distributed systems. To the programmer it should not matter where in the distributed system the actual objects are located, all that is required is a means by which operation invocations can be sent to the correct objects wherever they reside. Thus programming a distributed application should be no more complex than programming a centralised application providing that object location and access can be made transparent.

This paper describes the *Arjuna*<sup>19</sup> programming system currently under development at the University of Newcastle upon Tyne. Implemented in C++<sup>21</sup> *Arjuna* provides a number of flexible and integrated mechanisms for naming, invoking operations on (local or remote) objects, concurrency control, recovery control, object state management, etc. This flexibility is achieved by exploiting the inheritance capabilities of the implementation language.

## 2. Arjuna

While having similar aims to many other research projects (for example, *Emerald*,<sup>7</sup> *Clouds*,<sup>9</sup> *Avalon*,<sup>10</sup> *Argus*,<sup>15</sup> and *Camelot*<sup>20</sup>), *Arjuna* started with one major premise that differentiates it from them all in that the entire system had to be implementable using only standard compilers and systems. Thus it was decided at the outset that modifying a language or its compiler was not permissible. Since object-oriented languages seemed a natural starting point, the ready availability and portability of C++ was immediately appealing. The system that resulted comprises a stub generation and RPC system for distribution purposes, an object store for the storage of persistent objects, and a hierarchy of classes each of which contributes parts of the functionality required by the system as a whole.

Since the original design was conceived under version 1.1 of C++ the design was based upon the use of single inheritance only. Whether multiple inheritance would be an advantage or not remains open at this time. The decision to restrict the implementation of the system to standard compilers only, however, had a major impact in that it required that the programmer make explicit use of the inherited facilities.

The inheritance capabilities of the language provided a basis for the provision of basic capabilities to handle recovery, persistence and concurrency control to the programmer, while at the same time giving flexibility to the system by allowing those capabilities to be refined as required by the demands of the application. Thus the core class hierarchy of *Arjuna* appears to the programmer as the following:

- StateManager
  - LockManager
    - User-Defined Classes*
    - Lock
      - User-Defined Lock Classes*
      - AtomicAction
      - AbstractRecord
        - RecoveryRecord
        - LockRecord
        - and other management record types
- etc.

The following sections describe parts of the *Arjuna* system in more detail.

### 3. Stub Generation

Stub generation in *Arjuna* is different to that employed in other systems<sup>1, 5, 6, 12, 14</sup> in that it does not require the use of a separate Interface Description Language (IDL). Instead the *Arjuna* stub generator<sup>18</sup> is based upon the philosophy that the interface to an object has already been precisely specified (in C++) when the object was originally designed with a non-distributed implementation in mind. To this end the *Arjuna* stub generator accepts as input the standard C++ header files that would normally be input to the C++ compiler. This naturally enhances the transparency of the system in that the programmer need only produce a single object description as if the system was not distributed and rely on the stub generator to produce the distributed version automatically. Similarly, the programmer need not be concerned about mapping the C++ types used to the types supported by the IDL - in *Arjuna* they are the same.

Stub generation is not without its problems which principally stem from the lack of a shared address space between the caller and the actual object. Potential problems include those of:

- *Machine Heterogeneity.* Different machines may have different representations of various primitive data types. For example, byte ordering, arithmetic precision, etc.
- *Parameter Semantics and Types.* Stub generation usually utilises a copy-in, copy-out style of parameter passing which does not necessarily match the semantics of the local parameter passing semantics. Furthermore, certain types of arguments may be disallowed, for example, procedures.
- *Self-Referential Structures.* Linked data structures (which may even be circular) are an obvious source of potential errors
- *Failures.* Failure of an RPC is far more problematical to handle than failure of a local procedure call since the latter typically only occurs when the entire program fails or the error is expected. A procedure executed remotely can fail completely independently of the caller in unexpected ways.

The *Arjuna* stub generator attempts to compensate for these problems as far as it can automatically but there are cases where assistance from the programmer is required. For example, heterogeneity is handled by converting all primitive types to a standard format understood by both caller and receiver.



### 3.1. RPC System Interface

The RPC mechanism used in *Arjuna* (a multicasting version of *Rajdoot*<sup>16</sup>) is designed for general purpose use, and is thus not language specific. As a consequence it requires the programmer to convert and pack all arguments and results for a call explicitly into the buffers used by the RPC mechanism. Furthermore it requires clients to provide binding information which details the location of the remote service. Its primary primitives are:

- *Initiate*. This establishes a binding between the client and the server through the use of a manager process listening on a well-known port at the server's site. This manager process handles binding requests by forking an appropriate process and passing it the communication port to allow it to reply to the client. The server process then creates a new port and returns it to the client establishing a direct client-server connection. The manager process is not involved beyond the creation of the server.
- *Terminate*. This primitive terminates the binding between a client and a server and kills the server process.
- *Call*. This primitive performs the actual RPC. Its main parameters are an opcode indicating which procedure to invoke in the server and a buffer for the call arguments.

The actual interface to the underlying RPC mechanism is provided in *Arjuna* via the classes *Client\_rpc*, *Server\_rpc* and *rpc* which are C++ interfaces to the underlying RPC mechanism. The precise implementation details of these classes are irrelevant to this paper and so will not be presented here. What is important, however, is the interface they provide to the client and server stubs. The class *rpc* shown below

```
class RPC
{
    ...           // RPC opcode, buffers, etc.
};

class rpc : public RPC
{
public:
    rpc(long = 0);
    rpc(long, Buffer*);
    ~rpc();

    ...           // several useful operations
};
```

is the primary interface to the underlying RPC mechanism. Instances of this class are created and manipulated by both client and server stubs and are passed through the C++ interface to the RPC mechanism. In fact, the stub generator actually interfaces to the system via the classes *ClientAction\_rpc* and *ServerAction\_rpc*. These classes are derived from *Client\_rpc* and *Server\_rpc* and provide identical interfaces to those classes. They exist to ensure that any atomic action management information is correctly propagated between client and server.

This separation of the details of the actual RPC from the interface seen by the generated stub code is important and has many advantages. In particular, stubs can be generated without regard for the actual RPC mechanism used providing that the RPC mechanism complies with the required interface specification. In fact, as a simple experiment the underlying RPC system was replaced by the SUN RPC<sup>2,3</sup> mechanism without requiring any change to the stub generator or its output.

### 3.2. Remote server creation

The RPC mechanism requires an explicit call on its *Initiate* and *Terminate* primitives to create and destroy a remote object server and bind it to the client. The problem is when to perform these operations. Fortunately C++ provides a neat solution through the provision of constructor and destructor operations which are normally used for object initialisation and destruction. For remote objects the stub generator creates constructor and destructor operations that call the underlying RPC primitives. The resulting sequence of events then becomes:

- The (stub) object comes into scope in the application program and automatically invokes the object constructor.
- The stub generated constructor binds client to server using *Initiate*.
- A normal RPC is then made to invoke the real constructor for the object in the server passing any arguments supplied to the stub object constructor. This ensures that the remote object has been constructed at the same point in the application as the stub object and utilising the same arguments.
- All further operations are made as RPCs as operations are invoked in the client.
- When the stub object goes out of scope in the application, the generated destructor operation invokes the real destructor in the server via RPC before cleaning up the RPC connection using *Terminate*. This ensures that the real object is destroyed when the stub object is destroyed.

### 3.3. Parameter marshalling

Implementing remote procedure calls inevitably requires a mechanism by which arguments and results can be transferred between the client and the server. This typically involves packing the arguments into a buffer used by the underlying RPC transport mechanism for transmission and then unpacking them again at the receiving machine. These operations are frequently referred to as marshalling and unmarshalling.

Since the input to the stub generator is C++, there are several distinct kinds of parameters that may need marshalling:

- *Basic Types*. These are the standard built in types such as *char*, *int*, *float*, *unsigned*, etc.
- *Aggregate Types*. Arrays of other types.
- *Classes*. These are the fundamental units of abstraction and encapsulation in C++ and are the primary types for which stub generation is required.

In *Arjuna*, the C++ interface to the RPC mechanism uses an instance of the class *Buffer* (shown below)

```

class Buffer
{
    char *buffer_start;
    ...
public:
    Buffer (long);
    ...
    boolean pack (char);
    boolean pack (int);
    boolean pack (double);
    boolean pack (char*);
    ...
};

```

for parameter and result transmission purposes. This class provides overloaded pack and unpack operations that allow simple types to be marshalled and unmarshalled to and from the buffer with ease. Using this class an individual variable of basic type (say *int*) can be marshalled via the simple statement:

```

BufferInstance.pack(variable);

```

Similar statements can be generated to marshall arrays, etc., however, complications arise in marshallng and unmarshalling instances of classes. Recall that class instances are assumed to be encapsulated entities, thus only member or friend functions can access the internal state variables. The stub generator, therefore, augments the operations of any class with the two additional public member functions:

```

virtual void marshall (Buffer&);
virtual void unmarshall (Buffer&);

```

In order to avoid having to generate different code to marshall variables of different types the stub generator exploits the operator overloading capabilities of the language. Traditionally in C++, the operators '<<' and '>>' are overloaded to allow input and output of variables, including instances of classes (given suitable definitions of these operations by the programmer). The stub generator makes use of precisely the same technique, but defines the same operators to be the equivalent of marshall (<<) and unmarshall (>>) when applied to RPC buffers. The nett result of this is that variables of any type can always be marshalled using a statement of the form:

```

BufferInstance << variable

```

As an example consider the following simple class definition:

```

class Date
{
    // TRANSMISSIBLE

    int day, month, year;
public:
    Date (int, int, int);
    void set (int, int, int);
};

```

This class represents an object which will not be accessed via RPCs but may be sent as a parameter to some other remote object. When processed by the stub generator the resulting output header and the code generated for the marshall operation and redefinition of the operator << are shown below.

```
class Date
{
    int day, month, year;
public:
    Date (int, int, int);
    void set (int, int, int);
    virtual void marshall (Buffer&);
    virtual void unmarshall (Buffer&);
};

inline Buffer& operator<< ( Buffer& rpcbuff, Date& topack )
{
    topack.marshall(rpcbuff);
    return rpcbuff;
}

void Date::marshall ( Buffer& rpc_buff )
{
    rpc_buff << day;
    rpc_buff << month;
    rpc_buff << year;
}
```

As can be seen, the public interface to the class is unchanged beyond the addition of the marshall operations and instances of this class behave as they would have if the stub generator had not been used.

### 3.4. Remote objects

The stub generator currently distinguishes between objects designed to be accessed remotely and those that are merely transmitted as arguments. Only in the former case is full stub code generation required (the latter only needs marshall code generation as shown in the previous section).

For remotely accessible objects code must be generated for all of the public operations supported by the object together with an appropriate server capable of decoding the incoming RPCs and dispatching them to the correct operation in the real object. Thus the stub generator produces two distinct code fragments - one for the client, the other for the server.

For example, consider the simple class definition given below:

```

#include "Date.h"

class Calendar
{
    Date today
    ...
public:
    Calendar ();
    ~Calendar ();

    void setdate (Date);
    void getdate (Date&);
    ...
};

```

This class allows the user to create and manipulate remote calendar instances (for the purpose of arranging a meeting perhaps). Running this class definition through the stub generator yields the following two class definitions

```

class CalendarClient
{
    ClientAction_rpc *clientclass_rpc;
    Group *clientclass_server;
public:
    CalendarClient (ClientAction_rpc * = 0);
    void setdate (class Date);
    void getdate (class Date &);
    ...
};

class CalendarServer
{
    Calendar *therealclass;
    rpc *Calendar_1002 (Calendar *, rpc *);
    rpc *Calendar_1003 (Calendar *, rpc *);
    rpc *setdate_1004 (Calendar *, rpc *);
    rpc *getdate_1005 (Calendar *, rpc *);
public:
    CalendarServer ();
    void Server (int, char **);
    rpc *Dispatch (Calendar *, rpc *);
};

```

As can be seen, the client class (CalendarClient) provides the same public interface as the original class but its instance variables and the implementation of all of the operations have changed. Suitable name mapping tricks played with the standard preprocessor ensure that the programmer can still use this class under its original name maintaining transparency.

The generated server class (CalendarServer) contains a pointer to the real object which is only assigned when the object is actually created and initialised when an RPC for the constructor arrives. Similarly the object is destroyed when the RPC for the destructor arrives. Sample code generated for the getdate routine is as follows:



```

void CalendarClient::getdate (class Date& Par_0)
{
    rpc *callrpc = 0;
    rpc *result = 0;
    int validresult = 0;

    Buffer& rpc_callbuff = * new Buffer();
    rpc_callbuff << Par_0;

    callrpc = new rpc(1005, &rpc_callbuff);
    result = clientclass_rpc->Call(clientclass_server, callrpc);

    if ((result != 0) && (result->get_opcode() == S_DONE))
    {
        Buffer& rpc_retbuff = *result!get_buffer();
        rpc_retbuffer >> Par_0;
    }

    if (callrpc != 0)
        delete callrpc;
    if (result != 0)
        delete result;
}

```

### 3.5. Failure handling

The handling of RPC failures is the major problem in stub generation. This problem is actually exacerbated by the use of C++ since return values may be arbitrary complex objects. Unfortunately, there is no automatic solution to this problem. The stub generator only knows that operations pass and return instances of particular types when invoked and relies on being able to initialise a return value by unmarshalling it from the RPC reply. If the RPC fails the stub generator has no way of automatically producing code to return an error instance of the return type since it has no knowledge of how to construct such an instance. Handling such failures then becomes the task of the programmer. One possible approach, however, is to extend the current mechanism utilised to control the actual stub generation process (via special comments in the class definition) to include details of special error versions of objects which can be returned if an RPC fails. The current implementation takes the cavalier attitude that doing nothing is the safest approach, thus if the RPC call fails output parameters remain unset and nothing is returned

## 4. Reliability

Reliability of an *Arjuna* application is based upon the use of the well known atomic action (atomic transaction) concept.<sup>13</sup> Atomic actions have the properties of

- *Failure Atomicity*. All of the operations that comprise the action complete successfully or none of them do.
- *Serialisability*. The concurrent execution of actions is equivalent to some serial order of execution
- *Permanence of Effect*. Once completed new system states produced by actions are not lost.

Actions are implemented in *Arjuna* by the class `AtomicAction` which provides operations that correspond to those that would be familiar to any database programmer - Begin, End, and Abort.

Instances of `AtomicAction` must be explicitly declared and used by the programmer as appropriate to the application. Instances of `AtomicAction` only control when certain events happen on objects, that is, if an atomic action aborts any objects manipulated within it must have their prior state restored and concurrency control information updated. In order to accomplish this `AtomicAction` instances maintain a list of instances of classes derived from `AbstractRecord`. Each of these classes manages a certain property, thus `RecoveryRecords` manage object recovery, `LockRecords` manage concurrency control information etc. It is thus sufficient for the operations of `AtomicAction` to run down this list at runtime invoking the appropriate operation (for example, `top_level_commit`) on each record instance. This record based approach provides complete flexibility in that new record types can be created as required (other record types currently handle persistence, distribution and object scope).

#### 4.1. Recovery and Persistence

The failure atomicity and permanence of effect properties of atomic actions require that objects be both recoverable and persistent. These capabilities are provided by the classes `StateManager` and `ObjectState`. `StateManager` is the root base class of the entire *Arjuna* class hierarchy. As such it provides naming capabilities (in the form of unique identifiers), operations for the activation and deactivation of persistent objects along with state-based recovery. Since recovery and persistence have complimentary requirements (the only difference being where information is stored and for what purpose) the system uses instances of `ObjectState` for both purposes. When not in use persistent objects are stored in a passive form in an object store as instances of the class `ObjectState`. When first used they are automatically activated by the system which results in the conversion of the object into its standard runtime form. Deactivation occurs when the top-level action commits at which time the object is again converted back into an `ObjectState` for storage in an object store. Instances of `ObjectState` maintained for recovery purposes are held entirely in memory. More precise implementation details can be found in reference (11).

Conversion of objects between their passive and active forms is controlled by the system but uses operations supplied by the programmer. These operations, `save_state` and `restore_state`, must be provided (this is enforced by the compiler in R2.0 by making them pure virtual functions) otherwise neither the recovery system nor the persistence mechanisms will function properly.

#### 4.2. Concurrency Control

Concurrency control in *Arjuna* is the heavyweight control required for the provision of the serialisability property of atomic transactions.<sup>4,17</sup> It is not the lightweight control supplied by such constructs as semaphores or monitors. Concurrency control is provided by the system in the form of strict two-phase locking which by default supports a traditional multiple reader, single writer policy. The concurrency controller is implemented by a class derived from `StateManager` (the class `LockManager`) in conjunction with the class `Lock`. User defined objects are derived from `LockManager` and thus inherit concurrency control capabilities from it, and recovery, naming and persistence capabilities through it from `StateManager`.

`LockManager` is solely responsible for managing requests to set a lock on an object or release a lock as appropriate. It deliberately has no knowledge of the semantics of the actual policy by which the lock requests are granted. Such information is maintained by the `Lock` class instances which provide operations by which `LockManager` can determine if two locks

conflict or not. This separation is important in that it allows the programmer to derive new lock types from the basic Lock class and by providing appropriate definitions of the conflict operations enhanced levels of concurrency are possible.

## 5. Exploiting C++ Syntax

One criticism that could be levelled at the original *Arjuna* system is that it required explicit usage by the programmer to be effective. That is, the programmer must declare and use atomic actions appropriately, and make correct calls to the concurrency controller at the appropriate points in the code for an operation. Neglecting to do so was likely to lead to some potentially highly undesirable results. Thus, while the system provided great flexibility it also required careful use to avoid potential problems.

This criticism stemmed from the original design decision to use standard 'off-the-shelf' components in building the system. From the outset there was never any intention to either modify the C++ language itself or add features into any particular compiler. *Arjuna* was designed to be portable to any system that had a C++ compiler (regardless of origin) and ran Unix. Thus it was impossible to determine whether any operation modified a particular object without help from the programmer, since determining this fact would require semantic analysis of the particular operation.

However, with the release of R2.0 of the C++ language this criticism need no longer apply since it is possible to generate appropriate calls to the *Arjuna* system automatically as part of the stub generation process using only syntactic analysis of the signature of any particular operation. The key to this feature is the exploitation of the fact that operations can now be marked as *const* which implies that they do not modify the object to which they are applied. Utilising this syntactic marker allows the stub generator to insert calls to the concurrency controller to create read locks automatically when *const* operations are invoked, and to create write locks when non-*const* operations are invoked. In addition operations can have code generated that automatically creates and starts an atomic action when the operation is invoked, and commits or aborts it depending upon the result of the operation.

## 6. Retrospective

In retrospect C++ has proved to be an adequate language for programming a reliable distributed system like *Arjuna*. Where it has failed it has done so in part because of the actual implementation of the language by the currently available compilers and due to lack of certain facilities at the language level. In the following sub-sections these failures are examined in more detail.

### 6.1. Distribution and Stub Generation

The ultimate aim of the stub generation system is to take a standard C++ header file defining some class and produce from it an equivalent header containing a replacement stub-class definitions together with the code that implements the client and server operations of the stub classes.

This approach has several potential problems that arise due to a variety of causes. Firstly, typical C++ header files contain not only class definitions but also inline function definitions, macros, manifest constants, various preprocessor directives etc. Secondly, type information about parameters, etc. may be incomplete. For example, declaring a variable to be a pointer to some type does not imply that the variable points to a single instance of that type - it might actually point to an array (particularly if the type is *char \** - that is, a pointer to character which by convention is used to represent a string), or a list, etc. For this reason, the stub generator imposes certain semantics upon pointers such that a pointer to a type always points to



only one instance of that type - even for the *char* type. While this may seem to impose restrictions, classes can be written for strings etc. which make this restriction an inconvenience at worst.

Other problems arise due to the fact that, as was noted earlier, remote procedure calls typically provide a restricted set of capabilities over traditional calls. Thus the stub generator imposes the following restrictions upon the form of the class and the operations its supports:

- No public variables. The existence of public variables in a class definition breaks the fundamental encapsulation property assumed by the stub generator, so any such variables are removed automatically (with warnings) as the class definition is processed.
- Variable length argument lists to operations are disallowed since the stub code needs to know exactly the type and number of arguments to each operation for marshalling purposes. This restriction could be removed by allowing the programmer to specify a marshalling routine for such operations explicitly.
- Inline operation definitions are currently discarded as they make no sense to the stub generated version of the class at the client. Future versions of the stub generator will ensure that such inlines are made available in the server.
- Classes cannot contain static members since the stub generator cannot enforce the semantics of such variables correctly.
- Lack of support for operator overloading.

Some of these restrictions are due to weaknesses in the stub generator (for example, inlines and operator overloading) and will be removed in future versions. Others are fundamental to the technique of stub generation (static member variables) and will remain compromising the level of transparency achievable.

Using C++ as both the input and output language of the stub generator has had both benefits and problems. On the benefit side, application level location and access transparency is achieved due to the complete lack of a separate interface definition language. Similarly, exploitation of the operator overloading capabilities of the language in the output code has reduced the complexity of parameter marshalling considerably.

## 6.2. Reliability

Since atomic actions are not part of the actual implementation language there is a problem related to the scope of objects. Ideally the scope of an atomic action and of any objects it controls should be the same. However, it is easily possible to create programs where this is not true. For example by creating an action in an outer block, and an object in some inner block. In this case the object is automatically destroyed as the inner block is exited despite the fact that its ultimate fate has not yet been decided since the atomic action under whose control its operations have executed is still active. The *Arjuna* system attempts to compensate for this as much as possible by retaining object states until the outcome of the controlling action is known if the object is destroyed prematurely. Similar problems arise if objects are created and deleted on the heap while an atomic action is active. The problem then is how can the deletion of a heap object (and the associated freeing of memory) be undone if the action that did the delete aborts.

The lack of automatic garbage collection of objects is also problematical, since in a distributed system it is very easy to lose track of who is responsible for finally deleting a particular object, the nett result of which is objects are active longer than necessary tying up potentially valuable system resources.



Finally the lack of an exception handling mechanism remains a current problem (which may be eventually solved if the proposed exception handling mechanism is ever added to the language) which makes the transparent handling of RPC failures difficult if not impossible to implement.

### 6.3. Recovery and Persistence

The implementation of recovery and persistence is currently based upon the use of programmer supplied `save_state` and `restore_state` routines. Although the system uses R2.0 language features to force definition of these operations (pure virtual functions) errors in coding these routines can bring chaos. Such explicitly provided routines are needed because current compilers do not carry over type information about class member variables to execution time. If such information was available standard versions of `save_state` and `restore_state` could process it and obviate the usual need for the programmer to provide such routines. However, even then, there would still arise occasions where programmer assistance is required - particularly in the case where objects contain pointers to other objects, in order to determine precisely what constituted the state of any object. Similar techniques as those employed by the stub generator for producing the marshalling operations could be employed (that is `save_state` is almost equivalent to `marshall`) but that approach will never be completely automatic and will continue to require programmer assistance in several cases.

## 7. Conclusions

Object-oriented programming languages and systems still provide an ideal starting point for programming distributed applications and operating systems (for example, *Choices*<sup>8</sup>) due to their inherent modularity and encapsulation properties. However, using standardly available implementations of such systems may not provide sufficient support upon which to build distributed applications. It is in this area that C++ fails principally - current implementations are too biased towards an shared memory model of execution. Instead what may be required is a new implementation of the language such that the compiler can provide extra information not normally provided by the standard implementations - for example additional information about types and how they are laid out in memory, etc. If such a compiler existed and the proposed language extensions for exception handling, etc., were implemented within it, then C++ would be a far more realistic vehicle for implementing a reliable distributed system than it currently is.

## 8. Acknowledgments

The work reported here has been supported in part by grants from the UK Science and Engineering research Council and ESPRIT project No. 2267 (Integrated Systems Architecture).

## References

1. Sun Microsystems Inc., "Rpcgen Programming Guide," in *Network Programming Manual*, 1988.
2. Sun Microsystems Inc., "External Data Representation Standard Protocol Specification," in *Network Programming Manual*, 1988.
3. Sun Microsystems Inc., "Remote Procedure Call Protocol Specification," in *Network Programming Manual*, 1988.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

5. B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz, "A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 8, pp. 880-894, August 1987.
6. A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39-59, January 1984.
7. A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 65-76, January 1987.
8. R. H. Campbell, V. Russo, and G. Johnston, "Choices: The Design of a Multiprocessor Operating System," *Proceedings of the USENIX C++ Workshop*, Santa Fe, November 1987.
9. P. Dasgupta, R.J. LeBlanc, and E. Spafford, "The Clouds Project: Designing and Implementing a Fault Tolerant Distributed Operating System," Technical Report GIT-ICS-85/29, Georgia Institute of Technology, 1985.
10. D. Detlefs, M.P. Herlihy, and J.M. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++," *IEEE Computer*, vol. 21, no. 12, pp. 57-69, December 1988.
11. G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheeler, "The Treatment of Persistent Objects in Arjuna," *The Computer Journal*, vol. 32, no. 4, pp. 323-332, August 1989. (also in *Proceedings of the Third European Conference on Object-Oriented Programming ECOOP89*, ed. S. Cook, pp. 169-189)
12. P. B. Gibbons, "A Stub Generator for Multilanguage RPC in Heterogeneous Environments," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 1, pp. 77-87, January 1987.
13. J. N. Gray, "Notes on Data Base Operating Systems," in *Operating Systems: An Advanced Course*, ed. R. Bayer, R. M. Graham and G. Seegmueller, pp. 393-481, Springer, 1978.
14. M. B. Jones, R. F. Rashid, and M. R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp. 225-235, January 1985.
15. B. Liskov, "Distributed Programming in Argus," *Communications of the ACM*, vol. 31, no. 3, pp. 300-312, March 1988.
16. F. Panzieri and S. K. Shrivastava, "Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing," *IEEE Transactions on Software Engineering*, vol. SE-14, no. 1, pp. 30-37, January 1988.
17. G. D. Parrington and S.K. Shrivastava, "Implementing Concurrency Control for Robust Object-Oriented Systems," *Proceedings of the Second European Conference on Object-Oriented Programming, ECOOP88*, pp. 233-249, Oslo, Norway, August 1988.
18. G. D. Parrington, "Distributed Programming in C++ via Stub Generation," Technical Report, Computing Laboratory, University of Newcastle upon Tyne, January 1990.
19. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, "An Overview of Arjuna: A Programming System for Reliable Distributed Computing," *IEEE Software*, to be published.
20. A. Z. Spector, R. Pausch, and G. Bruell, "Camelot: A Flexible, Distributed Transaction Processing System," *Proceedings of CompCon 88*, pp. 432-439, February 1988.
21. B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

# Designing Portable Application Frameworks for C++

Fergal Dearle, Glockenspiel Limited, Dublin, Ireland.

Glockenspiel Limited  
39 Lower Dominick Street  
Dublin 1  
Ireland  
Phone: 011 353 1 733166  
Fax: 011 353 1 733034

Glockenspiel Limited  
2 Haven Avenue  
Port Washington NY 11050  
USA  
Phone: 516 767 7839  
Fax: 516 767 9067

## Abstract:

Glockenspiel CommonView™ is an application framework for C++, which allows portability of code between various presentation systems like MS-Windows, OS/2 Presentation Manager, X/11 and Apple Macintosh. CommonView has evolved from a simple set of **Window** classes which implemented a single text filing application to a full application framework now being used by thousands of developers to develop a diverse range of applications from CAD to Comms, DTP to Databases.

This paper describes some of the design principles behind CommonView. It examines how CommonView's internals have been structured to provide a consistent programming interface across a range of platforms.

## Introduction

The original design goals of CommonView were: To provide, in a C++ class framework, a simpler and more abstract mechanism for programming Graphical User Interface (GUI) environments than existing C toolkits. Applications written using the framework should be 100% portable across the range of environments supported. The framework itself should be easily portable.

The first two goals were achieved from the outset: CommonView's classes and member functions have not changed significantly since their original design and 100% portability between four of the major GUI environments has been maintained. However, designing the framework code for easy portability proved to be more elusive and required a major re-engineering effort to achieve it.

This paper covers the major design concerns for those wishing to produce a similar framework and shows how a layered approach can assist both the portability and usability of the framework.

Application frameworks are intrinsically different to class libraries. A class library contains classes which just abstract some ideas but which do not provide facilities for adapting the functionality of those classes through inheritance. e.g. At&T's complex library. An application framework on the other hand is primarily a platform for onward inheritance.

## Selection of Classes

The most critical part of designing a class library or application framework in any object oriented programming (OOP) language, is choosing the classes and hierarchies which will comprise the framework. Yet this is precisely the part of the design process for which it is most difficult to legislate since there are no well established design methodologies for OOP.

The criteria we must use when selecting classes are usability, extensibility and pertinence to intended use - properties which are normally measurable only after a class has been produced and tested! The design of CommonView was relatively easy in this matter: we chose to represent the visual elements of the GUI as classes, such as windows, scroll bars and buttons, since these were the objects for which we intended to supply programming support. It is not always such an obvious decision.

Take, for example, the design of a network resource management system for application programmers. The system should be simple, easy to use and provide networked file management, database access and remote printing services to the programmer.

Suppose the task of designing and implementing this framework is given to the communications team in our organization because of their proven track record in this field. To the comms designers, used to blasting packets into pipes and out again, C++ provides the perfect means of encapsulating the intricacies of packet transmission: create a set of packet classes with virtual transmit and receive members. For the comms developer C++ provides an elegant solution to the age-old problem of managing packeted information streams: a file transfer is a set of packet transmissions and so is a network print request. So all network resource requests can be described as sets of network packet transmissions. Before long a framework of classes based on network packets and sets of packets is created.



Unfortunately, application programmers attempting to use such a framework to receive a file from a networked database would need to understand the intricacies of the network packet system. This is precisely what Glockenspiel set out to avoid. What was really needed for the application developer was a set of abstract classes such as NetworkFile, NetworkPrinter and a simple member function interface with which to manipulate them. Adopting a layered approach to the design would have satisfied both the needs of the application programmer and the comms developer.

### The Great Hierarchy Debate

The structure of class hierarchies has been the subject of much debate, with the main arguments falling into two camps:

- Those who advocate the use of a single hierarchy of classes, where all classes are derived from a common ancestor. This approach is adopted by Smalltalk and many existing C++ class libraries such as the NIH library (also known as the OOPS).
- Those who advocate the use of multiple peer hierarchies and classes which cooperate.

We chose the latter approach, considering it to be the more flexible and better suited for ensuring the portability of our code.

There are many advantages to deriving a framework's classes from a common ancestor. The thorny subject of an object's persistence is much simplified if a common base class contains the virtual store and retrieve methods required - Smalltalk's sets and collections would not be as easily implemented were it not for its common base Object class.

I believe that while this method is well suited to other OOP languages, it is not necessarily suited to C++ for several reasons. My main criticism against this method is that it leads to inefficiencies when applied in C++. The late binding mechanisms in C++ mean that large class hierarchies containing virtual members are wasteful of memory since each class type requires the use of a static representation of the virtual function table.

Other inefficiencies occur in the construction and destruction of objects. The construction of a simple data class, for example a `pair` class to represent two integers, should happen instantaneously and not require any function call overhead. This is easily achieved if the `pair` class is declared as a separate entity outside of any hierarchy and its constructor and access member functions are declared as inline. It is not possible if the class is embedded in a larger hierarchy which also causes the cascading of many base class constructors in the process.

## Peer Hierarchies

By peer hierarchies we mean a set of classes and class hierarchies which cooperate together to achieve a desired function. The individual class hierarchies are only as small or large as they need to be to achieve this end. A framework of classes built in this way does not become encumbered with any unnecessary processing overhead. Classes in the framework are only related in ways that are useful to its function. For instance a `Window` object knows how to use a `Rectangle` object and a `Brush` object to draw a filled rectangle on its canvas. The `Brush` object in turn knows how to use a `Bitmap` object to create patterned fills.

Using multiple peer hierarchies and small classes allows us more flexibility when extending the functionality of the framework. Growth can take place laterally by adding new complementary classes as well as by providing more descendants of the existing classes. This gives the application developer the scope of being able to use frameworks from a multitude of vendors to solve specific niche needs.

## Portability Problems

In its original incarnation all of `CommonView`'s classes were declared in the public header files for all to view. This led to problems on several fronts.

As suppliers of complete class frameworks to application developers we strive to maintain the model of Producers and Consumers where the class descriptions in the header files define the contract between us. Unfortunately when so much of the internal data and code of the framework is published in the headers, the terms of our contract become fuzzy.

A case which illustrates this point is the SetStyle member function in the CommonView Window class. SetStyle was a protected member function in the Microsoft Windows version of the framework. Its function was to mask some flags into a status variable prior to creating a Window. The flags were specific to MS-Windows and it was never really intended for public use. Soon we were receiving a rush of technical support calls claiming that there was a bug in SetStyle because it had no effect after the window was shown.

SetStyle has since been supplanted by member functions which replace its function portably across all platforms, but it still remains in the PM and Windows versions of the framework for backward compatibility with some of our original consumer's code.

Another problem with this method is purely a question of aesthetics. Differences between the data members required for each version of the code are reflected in the public headers. Take the following example.

```

class App
{
private:
    Bool status;
#ifdef MSWIND
    HANDLE hApp;
    HANDLE hPrevInst;
    npchar CmdLine;
    int Scmd;
#elif PM
    HANDLE hAnchor;
    HANDLE hMQ;
    HANDLE hHeap;
#elif MACOS
    EventContext * Current_Event_Context;
#elif X11
    XtAppContext AppCntx;
    Display * AppDisplay;
#endif
    pCVLink List_of_EventContexts;
    pCVLink List_of_Menus;
    pCVLink List_of_Controls;
#ifdef MSWIND
    App ( HANDLE, HANDLE, pchar, int );
#elif PM
    App ( HANDLE, HANDLE, HANDLE );
#elif MACOS
    App ( EventContext * );
#elif X11
    App ( XtAppContext, Display * );
#endif
protected:
    App ();
    Exec ();
    int Start ();
    public: void Quit ();
    void };

```

The `CommonView App` class declared above is used to provide an abstract view of an applications execution model in GUI systems. All the user of the class should need to know is when and how to use the `Start`, `Quit` and `Exec` member functions of the class. Ideally the classes function should be easily gleaned from browsing the class definition in the header file. Classes defined in this way are almost impossible to browse effectively in a header file.

## Static v Dynamic Binding

Application frameworks are ideally suited to being made into dynalink libraries (DLL's). Frameworks build to exploit DLL's can act as pluggable components for application developers. Using DLL's the implementation of a framework can be changed without affecting applications using it. Achieving this level of modularity requires the removal of all implementation dependencies from the application into the DLL.

Applications using frameworks constructed in the traditional manner will be source compatible between different versions of the framework. Because object sizes differ between versions, applications will require to be recompiled every time the implementation changes. This situation is obviously not ideal. We would like to be able to ship application executables which are independent of their framework version.

The key to portability of applications using a framework therefore is the use of a common header file which contains no implementation specific data or types. What we need is a mechanism which hides the implementation differences between versions of the framework code, and denies access to the implementation to the application programmers.

## The Layered Approach

To achieve these aims we have reengineered the framework into a layered structure. `CommonView` now consists of over 120 classes which implement an abstract view of the underlying presentation system. Only 60 of the classes are visible to the application developer. The other classes are used to provide the internal layering of the framework.

### Requester and Implementer classes

The main thrust of our reengineering work involved the introduction of what we call the requester/implementer method of implementation hiding. Using this model the requester classes form the public part of the framework. In the networking example these classes would be the `NetworkFile` and `NetworkPrinter` classes used by the application developer.



Each requester class is mirrored by an implementer class which is responsible for servicing requests on behalf of the Requester. All the environment specific code resides in the implementers member functions. Using this approach the only data that needs to be in the requester object is a pointer to the implementer object. All the platform specific data now resides in the implementer class, hidden from view.

The requester class definitions specify a much more precise contract between producers and consumers. They provide a vehicle for greater portability of the framework code since the requester code can be common to all versions.

The CommonView App class now becomes a much simpler and more legible entity.

```
class App
{
    App_Imp * pAppI;
public:
    App();
    void Start();
    void Quit();
    void Exec();
    ~App();
};
```

Requester classes need not be restricted to one implementer. They may have several. For example we are planning in a future release to refine the current Window implementer class into canvas and window management implementers. In this way the canvas implementer class can then be common between a future printer class and the existing window classes.

## Virtual Toolkits

We can choose to write the implementer's member functions as a mirror of the requester class. Thus each requester function dispatches directly onto an equivalent member functions in the implementer. A better way is to use the layering provided by the model to provide a lower level of functionality in the implementer members. The implementer functions can provide a virtual toolkit which abstracts the common elements of the target systems' application programming interface (API). For example, the TextPrint member function of Window, which prints text onto the window's canvas area, can be implemented as follows:

```
void Window :: TextPrint ( char * string, Point p )
{
    pW->GetDeviceContext();
    pW->SetOrigin( p );
    pW->PrintString( string );
}
```

In the above example `pW` points to the `Window` class's implementer. The `GetDeviceContext` member has an analogous function on all the target API's: In X Windows it associates a `Graphics Context` with a window. On the Apple Macintosh it associates a `GraphPort` with a window handle and on OS/2 Presentation Manager it associates a presentation space handle with a window handle. Similarly `SetOrigin` and `PrintString` have analogous functions in each system.

We chose to use the Requester/Implementer model to increase portability of client code, but we have discovered that there are many other benefits to this approach:

- **Compiler Independence**

A hidden advantage to using this model is that it provides us with a certain amount of compiler and language implementation independence. The requester portion of the framework can provide us with an easy binding layer to compiler implementations. By compiling the requester code with the target compiler we can overcome many of the differences between different vendors implementations of the language.

- **Language Independence**

This model can even provide us with independence from the C++ language itself. An implementation of the `CommonView` framework for NeWS could for instance have much of its implementer code written directly in PostScript.

- **Memory Management**

The requester classes, used mostly by the client application code, are small and efficient. Thus they can be easily embedded into other classes or constructed on the stack. Stack usage is minimized because the larger implementer objects are allocated on the heap. The implementer classes perform all complex memory management avoiding many bugs.

An opportunity exists to use an alternate storage mechanism for the implementer objects. The requester could contain a database key for the implementer rather than a pointer. The framework could be engineered to keep only the currently active objects in main store, demanding objects from the database only as they are required.

- **Distributed Applications**

The requester/implementer model can be further extended by designing it around a distributed client server mechanism. In this way the framework can be distributed across a network. For example an application running on a graphics workstation using CommonView for its user interface code, could use a matrix manipulation framework on a departmental supercomputer. A future version of the CommonView framework will be build to exploit this, allowing for example Macintosh or OS/2 workstations to act as display servers for multiuser UNIX computers.

### **Utility Classes**

The requester/ implementer model has only been used in CommonView where it is necessary to hide implementation differences. Many of the smaller classes have been left with normal declarations for efficiency's sake.

### **Privileged Service classes**

CommonView exploits the C++ protection mechanisms throughout. This can cause problems when data in several of the Implementer classes is required by the framework to execute a base windowing system function. We overcome this problem by the use of privileged service classes. These classes are friends of the Implementer classes and provide the means to violate the protection of any class by providing access functions to its private and protected data members.

These service classes consist mostly of inline member functions. They take as argument a pointer to an object and return a reference the data member of the object. Since the access members of the service classes are inline there is no extra overhead in accessing the data member from a pointer. The main advantage of this mechanism therefore is in localizing unprotected access of private class data.

## **No Panacea**

Unfortunately there is no way of producing a totally portable design which will work on every vendors API. Where appropriate in CommonView we have added a C++ wrapping to base presentation systems C structs and functions. While not providing us with portability of code this does provide us with a means of manipulating the base API's features which is less error prone than traditional C code. Since these classes are buried within the implementation they at least have no effect on the portability of application code.

## **Progressive De-Abstraction**

Using the requester/ implementer model involves a small amount of overhead. The overhead consists of the time taken to dereference the implementer pointer and the function call overhead in calling the implementer's member functions. For most frameworks this overhead is not significant. Efficiency can be improved by what we call progressive de-abstraction. By this we mean using the implementation hiding model during the design and development phases but removing hiding from time critical classes in the production code. The Hiding need not be removed totally from the class. If all the implementer member functions are made inline performance can be increased without compromising the design model.

## **Conclusion**

The design methods described in this paper can be used to produce robust and portable application frameworks for C++. Glockenspiel is using this approach to extend the CommonView framework and to design and implement new Common\* frameworks.

## **Acknowledgements.**

Helpful comments and suggestions were provided by Jonathan O'Connor, Stephen Dewhurst, John Rowley and John Carolan.

## **References.**

- [1] S. C. Dewhurst and K. Stark, "Programming in C++". Prentice Hall 1989.
- [2] J. Carolan, "Constructing Bullet Proof Classes". C++ At Work '89 Proceedings.
- [3] A. J. Palay et al, "The Andrew Toolkit-- An OverView". USENIX Conference Proceedings, Winter 1988. pp.9.



[4] A. Goldberg, and D. Robson, "Smalltalk-80: The Language and its Implementation". Addison-Wesley 1983.

[5] K. Gorlen, "An Object Oriented Class Library for C++ Programmers", Software: Practice and Experience, vol. 17, no. 12, December 1987, pp. 889.

[6] M. Geary, "Developing Applications with Common Source Code for Multiple Environmnets", Microsoft Systems Journal, vol 4, no 1, January 1989.

1. The first step is to identify the problem. This involves understanding the requirements and constraints of the system. It is important to have a clear understanding of what the system is supposed to do and what resources are available.

2. The second step is to design the system. This involves creating a plan for how the system will be built. This includes deciding on the architecture, the data structures, and the algorithms that will be used.

3. The third step is to implement the system. This involves writing the code that will run on the computer. It is important to test the code as you go along to make sure it is working correctly.

4. The fourth step is to maintain the system. This involves keeping the system up to date and fixing any problems that arise. It is important to have a plan for how the system will be maintained.

# FOG/C++: a Fragmented-Object Generator

Yvon Gourhant, Marc Shapiro

Institut National de Recherche en Informatique et en Automatique

INRIA, B.P. 105, 78153 Le Chesnay Cédex, France

tel.: +33 (1) 39-63-55-11, telex: 697 033 F

e-mail: gourhant@corto.inria.fr

March 9, 1990

## Abstract

We describe a language and a compiler for writing distributed applications and systems, structured with fragmented objects. The language is an extension of C++. The compiler statically ensures the encapsulation of fragmented objects, the type-safety of remote communication, and generates code for the common cases of remote communication and object migration. It interfaces to the SOS operating system via a set of predefined classes, which are easily portable to other operating systems.

## 1 Introduction

The object-oriented programming approach allows to organize large software systems. As such, it has enormous potential in distributed systems, which are notoriously complex. However, most object-oriented languages have no provision for distribution.

Some distributed object systems are realized with C++ [Stroustrup 1985], without modifying the compiler or defining a new language. For instance, the Choices [Campbell 1989] operating system family uses C++ class hierarchies to facilitate the construction of customized operating systems. Similarly, Arjuna [Dixon 1989] provides objects for structuring fault-tolerant applications: persistent objects, atomic actions and a hierarchy of locks. Unfortunately, the programming interface to such systems is complex, because C++ provides no support for distribution and no concurrency control.

On the other hand, some object-oriented distributed programming languages, such as Emerald [Hutchinson 1987], Orca [Bal 1989] or Gothic [Banâtre 1986] has been proposed, but they tend to hide distribution under automatic mechanisms. While this may be a useful abstraction for the unsophisticated programmer, it is unhelpful for programming the distributed system itself, or for fine-grain control of the distribution of objects, related to their semantics.

Within the SOR project at INRIA, we have identified a useful extension to the object-oriented approach in distributed systems: the *fragmented object*. A Fragmented Object (FO) is an object which representation is distributed. We have implemented a distributed

operating system called SOS [Shapiro 1989b], based on the fragmented object concept. This system offers basic abstractions on which the system itself is built.

In this paper, we describe FOG, a language and a compiler for writing fragmented objects. The FOG language is an extension to C++. The language satisfies the requirements of distributed applications: such as choice of communication protocol and migration policy. By representing major abstractions (e.g. communication protocols) with classes, FOG supports extensibility of the underlying mechanisms. For instance, new communication protocols can be defined and interfaced by FOG, without changing the language. Furthermore, the FOG compiler automatically provides information required by the system, taking this burden from the programmer. It generates standard C++ code and declarations.

In the next section, we will introduce the fragmented object concept. Then we will present the language and its compiler for writing fragmented objects.

## 2 Objects in SOS

### 2.1 SOS Elementary Objects

SOS is an object-oriented distributed operating system, with support for arbitrary user-defined objects. It provides object creation, localization, invocation, storage and migration for elementary objects, called *sosObjects*. An elementary object executes within a *context* (or virtual address space) and is known to the system by a system descriptor. Other objects, that don't need system management, are plain C++ objects.

### 2.2 Fragmented Objects

A Fragmented Object (FO) [Shapiro 1989b] is an object whose representation is distributed within several contexts, possibly located on different sites. The FO is a single abstraction covering the concept of stubs, replicated objects and functional distribution.

As for any object, a FO can be accessed only via its strongly-typed interface. It is implemented as a group of *fragments*, which are elementary objects. Each one represents the FO in its own context and exports a local public interface. Just as an object may access its own representation directly, bypassing its public interface, similarly a fragment may access the FO's internal representation. Fragments may communicate, via communication *channels* (see section 2.4). The communications between contexts are only possible inside a FO.

### 2.3 Migration

Any FO grows by spreading: a fragment may create and add a new fragment to the FO, and then migrate it to another context.

In general, a FO represents a resource or a service. To communicate with a FO, a client has to require the migration of a *proxy* of this FO, in its context. A proxy



[Shapiro 1986] is a fragment that represents the FO in the client's context. The client locally invokes the proxy's methods. This proxy can execute locally, or it can relay invocations to a server. A server has a *server stub* that receives cross-context invocations.

Fragments that receive migration requests from clients are called *providers*. When a provider receives a migration request, it selects or creates a proxy candidate, and initializes it.

The proxy constructor is in charge finding its code object [Shapiro 1989b] and setting up its communication channels (see next section). Then the system migrates the candidate, in the client context.

Any fragment can play the role of proxy, server, and provider.

A FO can offer a different interface and implementation (i.e. a different proxy type) for different clients of the service realized by the FO. The proxy is selected by the migration strategy, relative to the client migration request. The programmer can specify different locations of invocation processing: local, remote or replicated.

For instance, if a client requests access to a file from a "file provider", it will migrate a different proxy for the file, depending on the access mode. All of the file's proxies, the provider, and the servers, form a single FO. Different implementation strategies are possible: e.g. the proxy can cache the file data locally; the remote server ensures protection and consistency; the file can be replicated between multiple proxies and servers. These different policies can all be specified with FOG.

## 2.4 Communication

The SOS communication protocols [Makpangou 1988] are interfaced by several object types: messages, channels, and *invocation objects*.

A hierarchy of message types provides several qualities of services and several addressing modes (1-to-1, 1-to-N, 1-to-1 of N). A hierarchy of channel types encapsulate virtual connections between fragments. The channel types provide diverse communication protocols. The channel methods define the invocation protocols (e.g. RPC, call without reply, multicast). Finally, an invocation object encapsulates results of asynchronous or multiple invocations.

These classes are portable to other operating systems.

This structure encourages the application programmers to use the most adequate communication protocol.

A cross-context invocation is implemented by first allocating a message and marshaling arguments into it. The message is passed as argument to the invocation of a appropriate method of a channel owned by the calling proxy. The system relays the call to the other end of the channel: it invokes the server stub of the called object.

The server stub unmarshals arguments and invokes the target method, corresponding to the operation code. The reply is treated symmetrically. The call is terminated when the caller's method returns and the invocation object is freed.

### 3 The FOG language

For taking care of the common aspects of programming fragments, we have implemented a Fragmented Object Generator. It is in charge of migrating objects, setting up channels, marshaling/unmarshaling arguments, sending and receiving messages, and managing exceptions.

The FO Generator (FOG) facilitates building fragments spanning contexts.

We present the language using the Name Service (NS) of SOS, as an example. The NS is structured in the following manner: on each machine, there is a provider of the name service, exporting proxies (`nsProxy`) to local contexts. Each `nsProxy` represents the NS in its context. All NS providers and `nsProxies` form a single FO. Each NS provider represents the NS on its site. They cooperate to map symbolic names to internal SOS references, exchanging local information. In addition to the provider role, they play both the role of proxy and server.

After this presentation, we show the extensibility of our approach. Finally, we give more details about the syntax for passing arguments.

#### 3.1 Structuring a Fragmented Object

The description of a fragmented object class is an enumeration of the class names of its individual fragments. For instance, the declaration of a FO named `nameService`, composed of two fragment classes `nsProvider` and `nsProxy`, is represented on Figure 1<sup>1</sup> (the declaration of `nsProxy` is not shown):

The description of each fragment class is composed of (see Figure 1):

1. A *local part*, with members labeled **public:**, **private:** or **protected:**, identical to a plain C++ class declaration. This part lists the fields and methods of a fragment which are accessed locally. For instance, in Figure 1, the local part contains a single, private, field `ch`.
2. *server methods*. This is the procedural interface of the fragment, which can be remotely accessed from other fragments of the same FO. Its members are labeled with the keyword "**group:**" (a member may be in both the local part and the server part by labeling it twice, e.g. **group**, **private:**, as member `recv` in Figure 1). We support exceptions, declared with the keyword **raises**; when C++ eventually supports exceptions, we will adopt the standard syntax.

All of the server method declarations are collated by the FOG compiler into a single *server stub*, which receives a message, unmarshals it, invokes the appropriate method, marshals the return values or a possible exception into a message, and return it.

3. *Client invocation*. A local method can interface to a method of another fragment of the same FO, with the "!" syntax. For instance, in Figure 1, the local method

---

<sup>1</sup>C++ keywords are presented in bold fonts; FOG keywords are bold underlined.

```

group nameService { nsProvider, nsProxy };

class nsProvider : public someClass
{
private:
    multiChannel ch;
    ...
public:
    ...
    int send (String s) ! ch.multiRPC() ! recv(s) raises(anException)
    {
        for (int nbReply=0; getNextReply(); nbReply++);
        return nbReply;
    }
group, private:
    int recv (String) raises(anException);
export:
    sosObject* createProxy (String myName, long& myReference)
        : nsProxy (myName);
};

```

Figure 1: Simple declaration of a FO and a fragment

`send` is in fact a local interface to a remote method. A communication channel must be specified (`ch` in the example), and optionally, a method of the channel to invoke (RPC by default, `multiRPC` in the example). Optionally the programmer may specify the name and argument list of the remote method (if different from the local one): in Figure 1 the remote method `recv` is invoked. Code to collect or process replies may be included in curly brackets (the `for` loop and `return` in the example).

Client invocations are translated by FOG into code to marshal arguments into a message, send it (using the specified method of the specified channel object), await a reply, and unmarshal the results, or raise an exception.

4. A *provider* part. The *provider method* is labeled by the keyword "**export:**". This declares a method for creating proxies and the constructor of the proxy candidate. The arguments form the migration request. In Figure 1, the provider method is `createProxy`, the migration request has two arguments (`myName` and `myReference`), and only the first is passed to the constructor of the proxy (`nsProxy`).

More than one proxy constructor can be specified, separated by commas. Code (of the method `createProxy`) that creates or select proxies may be included in curly brackets.

The FOG compiler generates: code to marshal/unmarshal the migration request into a message; the default code of the provider method, if not already provided by the programmer; and the constructor of each proxy candidate. If more than one kind of proxies can be migrated, the programmer has to write the selection code in the provider method `createProxy`.

### 3.2 Extensibility

The programmer declares the channel methods to be used (for specifying invocation protocols) and the invocation object methods (for getting reply from asynchronous or multiple calls). But the programmer does not write the code of invocations.

He can specify arguments to pass to the message constructor, e.g. for particular addressing modes (see section 2.4). For instance, the syntax for passing the value `v1` to the method `invoke` of the channel `ch` and the value `v2` to the constructor of an implicit message looks like this :

```
send (int v1, int v2, ...) ! ch.invoke (v1) : (v2) ! recv (...);
```

New protocols can be defined, extending the interface of existing channels and messages, without modifying FOG. For instance, ISIS [Joseph 1988] multicast protocols channels could be implemented by passing an ordering argument to the message constructor. FOG finds the class description of the channel and the message associated to the method declaration. It generates the declaration of a message that will contain the marshaled arguments. This message is added to the message class hierarchy. It inherits the qualities of service from the message associated to the invocation protocol `invoke` of the channel `ch` (see section 4.1).



Some communication protocol available with specialized channel classes, within a FO, can implement distributed synchronization: for instance, an atomic channel, that encapsulates a distributed critical section algorithm, facilitates replication. No additional syntax is necessary for remote synchronization.

### 3.3 Argument passing

#### 3.3.1 Optimization of argument passing

For optimizing argument marshaling, FOG adds two new type specifiers, out and inout, to the formal argument C++ syntax. An out argument is passed by result; an inout argument by value/result; a **const** argument is passed by value.

#### 3.3.2 Size of vectors

A pointer in C (therefore in C++) has two meanings: either a variable size array (a vector) or a memory reference (a complex object as a graph). Pointers can also be used to return multiple results from an invocation. This can be seen as a single range vector. In this section, we present our solution for dealing with vectors. The complex objects are described in the next section.

In the fashion of SUN/RPCGEN for vectors in C, we have added and extended the "<size>" syntax to C++. The size is either

- a simple value; for instance:

```
write (int, char <20>, int);
```

- or an argument of the invocation, e.g.

```
write (int, char <len>, int);
```

- or a procedure (or a method of an argument), e.g.

```
write (int, char <strlen(s)>, int);
```

- or a local instance of the embedded class, e.g.

```
class list { int len; char p <len>; list* next; ...};
```

This approach allows objects to define their own semantics (shallow or deep copy), for passing them as arguments of invocation.

### 3.4 Complex objects

If an object contains a pointer to another, FOG will not transmit the transitive closure. Automatic transitive closure is inefficient. Consider for instance, passing the head of a list, of which only the first item is used. Actually, FOG provides no support for pointer management. The programmer has to deal with *permPtrs*.

A *permPtr* [Shapiro 1989a] is a relocatable pointer, implemented in library. A *permPtr* instance contains a typed effective pointer. During migration, data pointed by *permPtrs* are not loaded. *PermPtrs* provide a conversion operator which charges the data, if needed.

### 3.5 Dynamic linking

The SOR group had previously extended the C++ language with a new keyword **dynamic**, to mark classes and instances which can migrate, and to simplify dynamic linking. We now consider it was a mistake to change C++ itself. The **dynamic** extension, described in [Gautron 1987], is now handled by FOG instead.

### 3.6 Type checking

The FOG compiler checks for compatibility between declarations of each end of a channel: for each client procedure, its counterpart with the same name and signature, is searched among all server stubs in the same FO.

Our remote type checking is based on the C++ type checking. The C++ programmer is not constrained to do data encapsulation: any field may be public. However, only the currently-executing fragment should have access to its own channels, because inter-context protection is based on channels within a same FO. Therefore, FOG constrains the programmer to declare the channels **private** or **protected**.

However, this protection is not sufficient because C++ has some features that violates encapsulation, such as friends, casts, and untyped pointers. We support them for efficiency and convenience.

## 4 Code generation

After checking type-safety of channels and messages, FOG generates standard C++ code and declarations for communication and migration. It also generates the necessary C++ declarations (a header file).

### 4.1 Communication

The code generation for remote communication is similar to a traditional stub generation [Jones 1985]. The SOS kernel communication primitive is modeled after the V-system RPC. The code of the stubs generated by FOG is similar to the extension for Modula-2 by Almes [Almes 1986] for the V-system.

FOG interfaces to SOS via the class hierarchies described in section 2.4. In addition it reuses marshaling for all communication protocols, encapsulated by channel and message classes.

For instance, for the declaration:

```
multiChannel ch;  
void send (int calledObject, String s, int i) ! ch.multiRPC()  
        : (calledObject);
```

FOG finds the declaration of multiRPC in the channel class multiChannel:

```
class multiChannel : public simpleChannel  
{  
    ...  
    multiRPC (invokeMessage&, ...);  
};
```

The message generated is something like this:

```
class fooInvokeMessage : public invokeMessage  
{  
    String s;  
    int i;  
public:  
    fooInvokeMessage (int called, String ss, int ii)  
        : (called), s(ss), i(ii) {}  
    ...  
};
```

Finally, FOG manages invocation objects (see section 2.4) for asynchronous and multiple calls. For instance, the following code iterates until a positive reply is received:

```
void nsProxy::lookup (const String s, out long& reference)  
    ! ch.multiRPC()  
{  
    while (getNextReply (reference) == 0);  
}
```

FOG generates:

```
void nsProxy::lookup (const String s, long& reference)  
{  
    ...  
    invocation inv = ch.multiRPC (...);  
    while (inv.getNextReply (reference) == 0);  
}
```

Heterogeneity is an important problem. It is not in the aims of SOS to support it. Nevertheless, the proxy principle makes support for heterogeneity quite natural. FOG is able to encode arguments in an external representation, such as XDR [Sun 1985]. It generates procedures for complex types, by decomposition in simple types. We do not provide details because heterogeneity is not the purpose of this paper.

## 4.2 Migration

For provider objects, FOG generates the code of the method for exporting proxies (see section 3), as well as the constructor of proxies. This code is very fastidious to write manually. The simplest migration represents about 50 lines.

Unfortunately, this code is dependent of SOS. In particular, it invokes the methods of *sosObject*, the base class of all objects managed by SOS.

The method for exporting proxies:

- unmarshals the importation request arguments (similarly to an invocation),
- ensures the FO's protection, by allocating a unique identifier for the FO (it invokes a method of the SOS object manager),
- invokes the method of creation or selection of a proxy candidate.  
If this method is not written by the programmer, FOG generates a default, which creates a single proxy and delegates a copy for every migration request. The programmer can provide alternative code, e.g. to migrate the candidate itself, a copy of itself, or to create a new proxy afresh each time.

Our generated proxy constructor finds and sets its code reference and allocates its channels.

## 5 Parameter passing

The RPC raises the well-known problem [Tanenbaum 1985] of the lack of transparency when passing parameters by reference: in cross-context invocations, all parameters are passed by value. To allow transparency with C++ local invocations, FOG offers two modes for passing arguments, according to the C++ syntax.

In the call-by-value mode, FOG does not form the transitive closure for pointers of complex objects, i.e. it does not follow pointers. The size given by the "<>" syntax is computed at compile time whenever possible. Otherwise, the code produced by FOG will compute the size at run-time.

Call-by-reference has different semantics, according to whether the argument is a shared object<sup>2</sup> or not. If this is not a shared object, FOG generates an equivalent call by value/result.

---

<sup>2</sup>A shared object is an object which can be accessed concurrently by several tasks.



Otherwise, FOG tries to migrate the objects passed by reference into the callee's context. We assume that the migration code (written by the programmer) already deals with the problems of sharing and synchronization. It must decide to: either migrate itself, or a copy, or delegate a proxy.

## 6 Conclusion

We have described FOG, a language and its compiler for writing fragmented objects; this is a good support for structuring distributed applications and the operating system itself.

For instance, we have presented the Name Service of SOS, implemented by a fragmented object. The remote communication between NS providers, for propagating changes, is encapsulated by typed communication channels.

FOG does not offer full distribution transparency, but high level abstractions to interface the complex system mechanisms, via a set of class hierarchies.

The interface to communication protocols is extensible and easily portable to other operating systems.

FOG also allows to create FO's by controlled migration. Unfortunately, object migration interface is dependent of SOS mechanisms. The success of our implementation is dependent of the **virtual** mechanism of C++.

C++ is a good platform for extension with distributed features. However, some features of C++ that are not object-oriented represent protection loopholes. A fragmented objects encapsulates protection domain spanning contexts, but C++ bad features are error prone.

We plan to add syntax for local concurrency control. This is difficult because concurrency should be orthogonal to inheritance and encapsulation [Kafura 1989].

## References

- [Almes 1986] Guy T. Almes. The impact of language and system on remote procedure call design. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 414-421, Cambridge, Mass. (USA), May 1986. IEEE.
- [Bal 1989] Henri E. Bal and M. Frans Kaashoek. Experience with distributed programming in Orca. Technical Report IR-200, Dept. of Math. and Computer Science, Amsterdam (The Netherlands), September 1989.
- [Banâtre 1986] Jean-Pierre Banâtre, Michel Banâtre, and Florimond Ployette. An overview of the Gothic distributed operating system. Rapport de recherche 504, INRIA, March 1986.
- [Campbell 1989] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of object-oriented operating system design. Technical Report R-89-1510, Department of Computer Science, University of Illinois, Urbana, Illinois (USA), April 1989.
- [Dixon 1989] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheeler. The treatment of persistent objects in Arjuna. In *ECOOP '89*, Nottingham (GB), July 1989.

- [Gautron 1987] Philippe Gautron and Marc Shapiro. Two extensions to C++: A dynamic link editor and inner data. In *Proceeding and additional papers, C++ Workshop*, Berkeley, CA (USA), November 1987. USENIX.
- [Jones 1985] M. B. Jones, R. F. Rashid, and M. R. Thomson. Matchmaker: an interface specification language for distributed processing. In *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*, pages 225–235, New Orleans LA (USA), January 1985. ACM.
- [Joseph 1988] Thomas A. Joseph and Kenneth P. Birman. Reliable broadcast protocols. Technical Report TR 88–918, Dept. of Comp. Sc., Cornell University, Ithaca, New York (USA), June 1988.
- [Kafura 1989] Dennis G. Kafura and Keung Hae Lee. Inheritance in actor based concurrent object-oriented languages. In *ECOOP'89*, Nottingham (GB), July 1989.
- [Makpangou 1988] Mésaac Makpangou and Marc Shapiro. The SOS object-oriented communication service. In *Proc. 9th Int. Conf. on Computer Communication*, Tel Aviv (Israel), October–November 1988.
- [Hutchinson 1987] Hutchinson Norman C, Rajendra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald programming language. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, WA (USA), October 1987.
- [Shapiro 1986] Marc Shapiro. Structure and encapsulation in distributed systems: the Proxy Principle. In *Proc. 6th Intl. Conf. on Distributed Computing Systems*, pages 198–204, Cambridge, Mass. (USA), May 1986. IEEE.
- [Shapiro 1989a] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. Persistence and migration for C++ objects. In *ECOOP'89*, Nottingham (GB), July 1989.
- [Shapiro 1989b] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4), December 1989.
- [Stroustrup 1985] Bjarne Stroustrup. The C++ Programming Language. Number ISBN 0-201-12078-X. Addison Wesley, 1985.
- [Sun 1985] Sun Microsystems, Inc. External data representation reference manual. Report 800-1177-01, Sun Microsystems, Inc., January 1985.
- [Tanenbaum 1985] A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4):419–470, December 1985.

# Object-Oriented Redesign using C++: Experience with Parser Generators

Judith E. Grass, Chandra Kintala, Ravi Sethi

*AT&T Bell Laboratories, Murray Hill, New Jersey 07974*

Yacc is a popular parser generator that has been in use for over 15 years. It was a pioneering effort when it appeared. Advances in parser generation and Yacc's widespread use have created a demand for changes and enhancements to the original program. This paper contains some observations on object-oriented redesign based on our experience with implementing two parser generators, Kyacc and Ryacc, in C++. Kyacc is a compatible reimplement of Yacc, which we hope to use as a base for experimenting with variants on Yacc. Ryacc generates right-regular parsers; it was implemented by adapting the existing code for Yacc.

**Keywords:** C++, object-oriented design, prototyping, parser generators

## 1. Introduction

Johnson's popular parser generator Yacc [4] is well engineered, for its time. It was designed to cope with severe memory limitations, its implementation language C [5] was still under development, and the prevailing design techniques favored functional decomposition rather than object orientation. Years of performance tuning without substantial changes to the original design have resulted in a brittle program that is hard to understand and modify. Meanwhile, several alternative parser generators have been attempted, motivated by changes like the following: better error recovery; the use of newer look-ahead computation algorithms during parser generation; more semantic constructs such as symbolic *\$name* attributes, or perhaps attribute grammars; rules with regular expressions in the right-hand side; flexibility in the style of the generated parser, to allow table-driven or C++ based parsers; encapsulation of the generated parsers to allow multiple parsers to be used concurrently.

We approached a Yacc-like tool from two directions, leading to two distinct parser generators, *Kyacc* and *Ryacc*.

- *Kyacc* was designed to be a Yacc replacement that could serve as a base for experimenting with enhancements. It is a fully compatible reimplement; it retains the original efficiency of Yacc.
- *Ryacc* was designed to handle the more complex ELALR(1) grammars [6], with regular expressions on the right sides of productions. It is an adaptation of the existing Yacc code.

Kyacc and Ryacc are the results of object-oriented design techniques, as we understand them; in particular, of encapsulation and inheritance. Both are implemented in C++ [12], an efficient and widely available object-oriented extension of C. Data structures corresponding to parser objects such as nonterminals, productions, and states are naturally encapsulated into classes. These data structures take advantage of inheritance; for example, nonterminals and terminals inherit some operations from symbols.

The implementation of a Yacc-like parser generator is an interesting case study in object-oriented design and programming. Although Yacc's code is relatively small in size, its functionality is complex, it has rich data structures, it uses many clever implementation tricks, and, after years of tuning, it is compact and fast. Replacing such code with a clean object-oriented design and implementation, without compromising performance, is a worthwhile challenge.

This paper describes some observations on object-oriented redesign and on C++ as an implementation language, based on our experience with Kyacc and Ryacc. In the next section, we describe the observable design process for Kyacc and Ryacc and emphasize the need for iteration during design. In Section 3, we list some C++ concepts, such as inheritance, inlining, and virtual functions, as applied to parser generators. We conclude with a summary discussion of these observations.

## 2. Issues During the Redesign

Design begins with an understanding of the problem. Before reimplementing Yacc as Kyacc, it was necessary to understand precisely what Yacc did. This task was quite complex and time consuming. For example, Yacc uses a fixed size array of integers, `mem0`, to store both productions and states. After the productions are read in, the remaining space is used to hold states. The code was hard to read, despite the following comment in the function that generates states:

“[this] represents the magic moment when the `mem0` array, which has been holding the productions, starts to hold item pointers, of a different type . . . ”

However complex Yacc's code may be, it is nevertheless a precise specification of the function and performance goals to be met by Kyacc.

The specification for Ryacc was not quite as precise, and many aspects of it were not fixed. Ryacc had to correctly build parsers for ELALR(1) grammars, but the way the grammar was specified and the way semantics and error recovery were to be handled was open to experiment. The open-endedness of the Ryacc specification meant that there was no complete overview of its specification to support a top-down design approach.

The initial design of these parsers involved identifying the objects and the functional phases. The basic objects for both parser generators include: grammar symbols, both nonterminals and tokens; a symbol table with operations for entering and looking up symbols; grammar rules; a complex finite state machine that was either an LR(1) parser or an ELALR(1) parser comprised of parser states and parser actions; and a tabular form of the parser machine that could be used as part of the final parser. Functional phases could not be entirely ignored here. Any parser generator has at least three major functional components [Figure 1] which: 1) scan and parse the user specification to build a collection of rules and grammar symbols; 2) build the parser (states and actions); and 3) generate and output tables for the final parser. These are distinct phases dictated by the basic algorithms for building parsers.

Parnas [10] states that “it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart.” He proposes instead that modules be used to



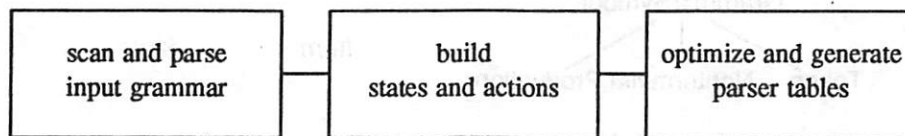


Figure 1. Functional components of a parser generator.

---

hide design decisions. We interpret these remarks as advocating design based on objects. Meyer's [7] more recent refinement of this process is:

- find the objects,
- describe the objects,
- describe the relations and commonalities between objects, and
- use the objects to structure the programs.

Although we began with the code for Yacc, a finished parser generator, the objects were not evident at the outset. These design steps were iterated, as we describe below. Wirth [13] states a similar need for iteration, "Often one discovers the right data structures only while developing the entire program. Without experience and foresight, one is condemned to restructure a program's module decomposition from time to time."

## 2.1. Finding the Objects

We began with an ambitious goal of designing an object-oriented program that could be decomposed along functional lines. This approach led to a class hierarchy where inheritance is used to partition the objects along functional lines; an excerpt appears in Figure 2. A set of base classes, *toka*, *nona*, *proda*, was created for grammar symbols, tokens, nonterminals, and productions, respectively. They contained the basic information about the grammar, such as the string representation, where they appeared in the grammar, etc. A new set of classes, *tokb*, *nonb*, *prodb*, was derived from the previous set to add functionality and some of the information gathered during grammar analysis. Yet another set of classes, *tokc*, *nonc*, *prodc*, was derived to add information specific to LALR parsing. The classes in the first two sets could be reused in any other, e.g., top-down, method of parser generation as well.

We abandoned the first design because the objects were not clear at the outset, much less their functional partitioning. Smalltalk [2] was used to gain experience with objects appropri-

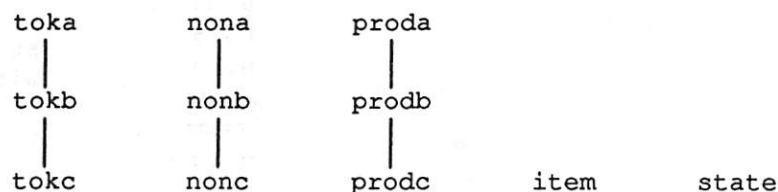
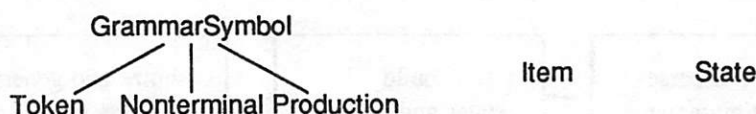


Figure 2. Class inheritance in Prototype 1 of Kyacc.

---



**Figure 3.** Class relationships in (Smalltalk) Prototype 2 of Kyacc.

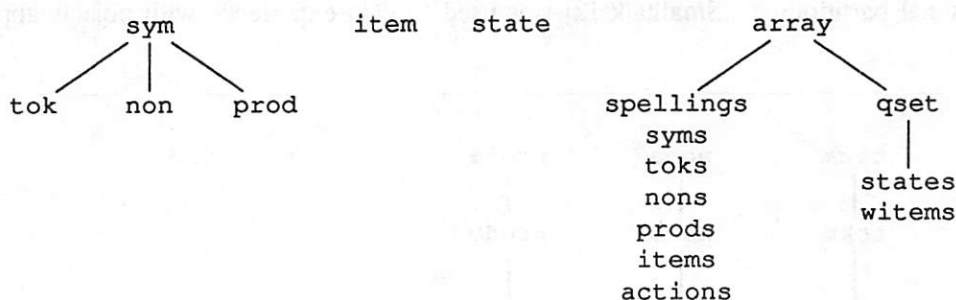
---

ate for LALR parsing. Its rich set of built-in classes, Dictionary, Set, OrderedCollection, and their methods facilitated rapid experimentation with different designs for the object classes. A simplified view of the Smalltalk class hierarchy for Kyacc appears in Figure 3. For a flavor of this prototype, see the grammar manipulation program in Section 6.4 of Sethi [11].

We used the design of the class structures developed in Smalltalk to implement the final prototype in C++. The Smalltalk class design remained essentially intact till the end. The classes and their inheritance relationships in the final C++ implementation appear in Figure 4.

Ryacc was not prototyped in the same way as Kyacc was. Ryacc was built in phases. We began by rewriting a copy of Yacc so that it was compilable by C++ and by reorganizing some of the code to make it easier to understand and manipulate. The first phase of Ryacc's development was driven by the basic flow of parser generation as shown in Figure 1. At first we kept close to Yacc's basic structure, but as it became clear just how different the problem of ELALR parser generation was, the divergence became pronounced. The first working version of Ryacc used classes as a data encapsulation mechanism, but made no use of inheritance.

The algorithms used to generate an ELALR(1) parser from a right-regular grammar are expressed in terms of graph manipulations [6], [8]. Symbols, terminal and nonterminal, appear mainly as members of sets labeling transitions in the graphs. The main design effort in Ryacc went into the design of a reusable set class and the design of graph classes. The class `sym` represents both terminal and nonterminal symbols and Ryacc regular expression operators. Terminal and nonterminal symbols are entered into a name table `name_tab` and assigned unique hash numbers. All terminals are chained together, and all nonterminals chained separately. This makes it easy to iterate through these two kinds of symbols. The definition of terminal and nonterminal symbols could have been refined by inheriting from a common sym-



**Figure 4.** Class relationships in the final (C++) Prototype of Kyacc

---

bol base, but since our algorithms do not frequently need to make a distinction, collapsing them was convenient.

It became apparent as more functionality was added to Ryacc that some of its structures had deep similarities and that inheritance could be exploited to clean up the design. Redesign was accomplished by extracting the old code for a component and replacing it with the code for the new version of that component. Generally this was not as painful as it might sound. The most primitive Ryacc design was built around encapsulation. Most redesign did not alter the interface between components, so changes could be localized. The integer set class proved so useful that it was spun off as a separate reusable library. The development of Ryacc was an iterative process starting with a rather crude design, but which developed into a more refined design as practical aspects of ELALR parser theory became better understood.

## 2.2. The Object Relationships

Inheritance is a way of further abstracting data: common aspects of a whole family of data kinds are factored into the base class. Inheritance also allows an object to be implemented by adaptation from an existing base class without having to reimplement the services it needs from the base class. This facilitates a great deal of reuse.

For example, Yacc uses integers to denote tokens and nonterminals and uses a boundary value, NTBASE, to distinguish them in productions which are stored as arrays of integers separated by negative production numbers. The corresponding base class in Kyacc is `sym`, from which are derived classes for tokens, nonterminals, and productions. Productions are stored as sequences of `sym` object pointers, as in Figure 5. Thus, each production takes up a segment from that array. This use of inheritance provides a better structured and more readable program, without losing the original efficiency of having a single array of integers for all the productions.

Kyacc has another base class, `array`, from which other arrays are derived, including `toks` and `nons`. Class `array` has not only data items such as name and size but also methods for getting the next element, for reporting errors, etc. In Yacc, this inheritance is implicit in the macros for manipulating those arrays.

There is also another form of inheritance in parser generators that we call *algorithmic inheritance*. Several stages in Yacc compute sets by making repeated passes until a stability criterion is met; such sets are implemented as arrays. For example, during state generation,

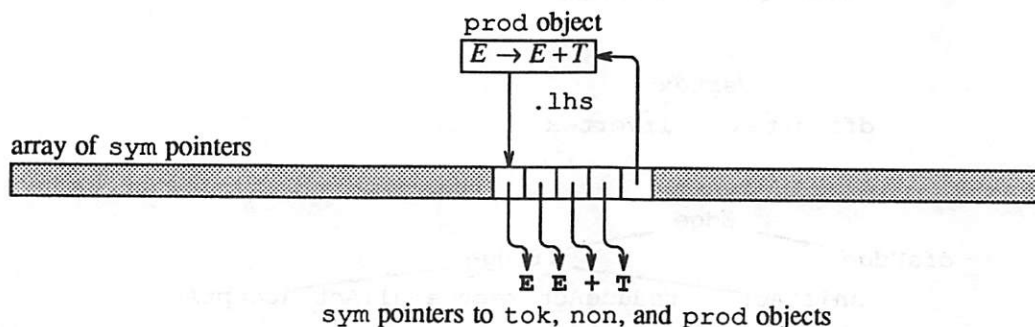


Figure 5. The syms array in Kyacc.

Yacc starts with an initial state in the state array. Examination of each state in the array can result in new states to be added. If a new state is already present in the array, it and all the following states in the array may need to be reexamined. This process of repeated passes over an array is facilitated through a class called `qset`. From `qset` are derived class states for generating states and class `witems` for computing the closure of items belonging to a state.

Finally, the parsing actions in a parser generator are of 3 kinds: shifts, gotos, and reduces. Yacc uses different integers to store different types of actions on the states and keeps a single integer array, `amem`, to store all the actions for all the states. In Ryacc, it is quite natural to have a base class called `action`, from which classes `shift`, `goto`, and `reduce` are derived. As in the array `syms` for productions, all the actions for all the states are stored in a single array of pointers to action objects. However, reiterating over the design process revealed that reduce actions are fundamentally different from the other actions; so they were separated out into a different class and a separate array was created for them.

### 2.3. Inheritance in Ryacc

The basic algorithm to build ELALR(1) parsers requires the rules to be in the form of a deterministic finite automata (DFA). The transitions of the rule DFAs are labeled with sets of symbols. These automata are fundamental objects in Ryacc that are not present in Yacc. There does not seem to be any straight-forward or "user friendly" way to directly give a textual specification of these DFA, especially when one of the design goals of Ryacc was to retain the "look and feel" of Yacc. The dual representation of rules as syntax trees and DFAs appears unavoidable in this context.

The ELALR(1) parser itself is a large graph with vertices defined by the states of the parser and edges defined by the state transitions of the parser. The parser graph is built by a complex conversion from the entire set of rule DFAs and symbol information.

The rule graphs and parser graph were implemented using inheritance from a common base graph type. Each kind of graph is made of three components: a root node, vertices, and edges. The root of a graph contains a list of vertices. Each vertex contains a list of edges. The rule DFAs and the parser automaton represent different specializations of this basic structure, as in Figure 6.

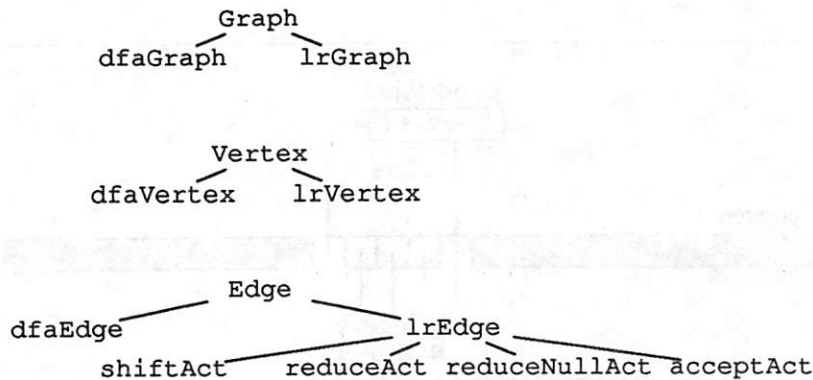


Figure 6. Graph class relationships in Ryacc

---



The basic graph classes, `Graph`, `Edge`, `Vertex`, contain functions to add states (e.g. vertices) and transitions (e.g. edges) as well as to locate states and edges from the symbol sets that define them. The rule graph classes inherit from these using the components `dfaGraph`, `dfaVertex`, and `dfaEdge`. They contain more attributes than the basic graph classes and contain functions to construct a graph from an expression tree and to optimize a graph.

The most elaborate set of graph classes represents the ELALR(1) parser. The basic parser graph inherits from the base graph class to create the classes `lrGraph`, `lrVertex`, and `lrEdge`. Derived class `lrGraph` adds virtually no attributes to the base `Graph` class, but its constructor builds the ELALR(1) parser graph from the set of rules, builds tables from itself, packs the tables and prints them. It is the driver for most of the program. Class `lrVertex` represents a parser state. Its attributes include an integer set representing the set of parser items defining the kernel of the state, another set representing the nonkernel items and two additional lists of parser transitions used to build parser tables. The `lrEdge` class represents parser transitions. These contain the information about all of the parser actions. An ELALR(1) parser uses more kinds of actions than an LALR(1) parser such as Yacc. There are two kinds of shift actions: one that shifts a symbol and marks the parse stack, and one that shifts a symbol with no marking. There is no need to distinguish between the shift of a terminal symbol and a nonterminal symbol before table packing. There are four kinds of reduce actions: a simple reduction which uses a stack marker to find the handle, a null reduction, an accept action, and a reduction using a *readback machine* to locate the handle. These actions are differentiated by deriving each as a new class from the `lrEdge` class. The two kinds of shift actions are collapsed together as are the simple reduce action and the reduce with readback. This proved to be more convenient than separating them. Inheritance and the use of virtual functions for the actions led to very clean code where actions were scanned, printed or packed into tables.

### 3. Implementing in C++

#### 3.1. Inheritance to Simulate Genericity

As in Figure 4, class `array` is used to derive a number of arrays of different types, including `syms`, `toks`, `nons`, `items` in `Kyacc`. Class `array` has protected members for the name, size, and limit, and a `getnext()` function. The derived arrays implement type-dependent operations such as `push()`, overloaded operator `[]`, `find()`. Although this use of inheritance to simulate generics in `Kyacc` is inelegant and increases source code size, it has actually been beneficial to have an efficient type-dependent implementation of some generic routines. For example, `toks.push()` pushes a pointer to a character string and initializes the token with its precedence and other information, whereas `prods.push()` merely increases the current size of that array. The encapsulation and inheritance mechanisms of C++ allow uniform naming of generic operations, with the variant implementations hidden in the derived classes.

A case could be made that `Ryacc`'s graph classes also use inheritance to simulate genericity. Although the contents of vertices and edges differ in the DFA graphs and the parser graph, they otherwise behave more or less alike. The use of a generic graph class would make it possible to tie together the components of the DFA graph and the parser graph in a way that is not possible using inheritance.

The most blatant example of using inheritance to simulate genericity can be found in the code for the `Ryacc` parser skeleton. The parser implements an automatic error recovery mechanism [1] that uses a deferred parsing strategy. This implies multiple parse stacks. The parser in effect keeps two parse states: an advanced parse state that does not reflect the effect

of reductions, and a parse state that lags a few tokens behind and maintains calculated semantic values. The advanced parse is meant to catch and recover from syntactic errors before they can affect the semantic state of the lagging parse. These use a different version of a stack to maintain the state. A third kind of stack is constructed to act as a test area during error recovery. These three kinds of stack (`parse_stk`, `adv_stk`, `copy_stk`) are derived from a base class `stack`. The stack data in `stack` is defined as an array of `void*`. It implements the basic stack operations. The derived classes reimplement the basic operations with coercive routines of this sort:

```
sim_entry *pop () {return((sim_entry*) stack::pop()); }
```

This kind of trick will not work everywhere. We wanted to encapsulate the Ryacc parser as a class with no globally available data. This would make it easy to instantiate parser classes for several grammars in one program. The biggest obstacle to this is the language dependent symbol type. For error recovery, the parser needs to be able to create symbol objects. To do this the parse uses `new` and the name of the symbol class. Because we did not wish to implement the parser as one huge macro, there was no reasonable way to parameterize the name of the symbol class, so there does not appear to be a way to make this creation process generic.

### 3.2. Dynamic Binding, Virtuals and Inlining

We were not as much limited by the lack of dynamic binding in C++ as by the limits on how the virtuals can be used. In Kyacc, for example, each production is stored as a sequence of `syms`, where each `sym` is a `tok`, `non`, or `prod`. Ideally, `istok()`, `isnon()`, `isprod()` would be virtual functions in class `sym` returning `false`; any derived class would reimplement the corresponding function to return `true`. However, since virtual functions cannot be inlined in C++, such an implementation of these three functions would be inefficient. There are other instances in Kyacc where virtual functions provided dynamic binding in a clean fashion; for example, the `isnull()` function in class `sym`.

Both the DFA graph classes and parser graph classes in Ryacc contain a large number of access functions like the following member of class `lrVertex` (see Figure 6):

```
lrEdge *get_trans() { return ((lrEdge *) transitions); }
```

This returns the transition list for this LR state. Ideally, this could be a virtual function defined in the base class. Every kind of vertex defined from the base class `Vertex` needs to do this. However, the transitions of an `lrVertex` belong only to the class `lrEdge` as the transitions of a `dfaVertex` must belong only to the class `dfaEdge`. Since the `get_trans` function must return a different type in each case, it must be non-virtual. There does not appear to be any good way to specify that a collection of classes such as `lrEdge` and `lrVertex` exist only as components of another class `lrGraph`. As a result, it is necessary to use many explicit coercions of pointers to graph components and to implement a lot of these little coercive access functions.

### 3.3. Pointers, Objects and References

The lookahead set (`laset`) in an LALR(1) item in Kyacc is implemented as a pointer to an integer set in an array of int-sets. Each int-set is a bit vector of a certain size, dependent on the grammar. Since there is no guarantee of the initial values of integer objects, allocating and initializing them all at once in one array class of int-sets gives an efficient implementation. Class `set` can then hide the implementation details, such as an index into that array, and provide an external interface of readable and high level set operations. It also avoids having

explicit set objects inside the items and passing references around for set manipulation. The set-equality test is implemented as an overloaded operator= inside a set class and uses a reference to the second set as a parameter. However, the set union function is not implemented as an overloaded operator but as an ordinary function taking a pointer to the second set as a parameter. The body of the set-union function is almost identical to that of Yacc; thus, it achieves the same efficiency at the same time as it encapsulates the implementation details within the class.

Profiling helped greatly in quickly isolating functions that consumed large percentages of the total time. For example, profiling revealed that early implementations of Kyacc spent the first 80% of the time in a few set operations on lookahead sets in items. A change in the implementation of sets resulted in a Kyacc where the first 80% of the time is more broad-based. Encapsulation coupled with profiling provides an easy iterative process to tune the performance of a system productively.

The most difficult category of changes involved experiments with the use of the integer set class in Ryacc. Integer sets are used to define ELALR states and DFA transitions. They are used to label all shift actions. The process of creating new states and generating lookahead sets creates and destroys lots of them. We profiled Ryacc using gprof, and it became apparent that set constructors and operations were the primary performance bottle neck. A good deal of this was due to the construction of temporary sets for operations on set objects. Changes to the set class that minimized set copying were easy to make, as they could be localized. Experiments involving the use of set pointer members instead of set object members in the graph classes were more difficult. Classes manipulated their own set members directly, and tracking down all the references proved to be difficult, especially when these members were passed to other routines. Ultimately, these changes led to a faster parser generator, but they muddled the design.

### 3.4. Code Reuse

In designing Kyacc and Ryacc, we tried to reuse pieces of Yacc code as much as we could. It was a simple exercise to rewrite Yacc's function declarations so that they could be compiled with C++. Nonetheless, our reuse of Yacc code turned out to be almost entirely confined to the front end. The Yacc code for scanning and parsing a grammar specification was adaptable to the more complex syntax of right-regular grammars in Ryacc. However, while the result of parsing a Yacc rule is a simple list of symbols, the result of parsing an Ryacc rule is an expression tree. In a sense, Ryacc rules are three dimensional, and this fundamental difference made it impossible to borrow any more code from Yacc. Finally, even the parsing code taken from Yacc was abandoned. Ryacc's input language is LALR(1), so we used Yacc to generate a parser containing C++ semantic actions which we compiled using C++, version 2.0. The fact that Yacc can process semantics in C++ and can in turn be compiled with C++ is a happy coincidence. Yacc does little more to the semantic specifications than simple substitutions on tokens with an initial '\$' symbol. Everything else within balancing braces is copied verbatim.

Some aspects of Yacc's specification are clearly dictated by the limits of the C language and really should not be carried over into a C++ parser generation tool. One of the clearest examples of this is the use of a C union to carry the semantic values of symbols. The declaration of symbol types that are tied to the names of members of the YYSTYPE union gives only very weak capabilities for type checking. This is clearly the kind of application that begs to be implemented using a common symbol base type with derived classes to hold various semantic value kinds. The benefits of such an approach are discussed in [3]. Implementing this kind of semantic scheme is an area for further work in Ryacc and Kyacc.



### 3.5. Flexibility

Modularity is fundamental to design for modifiability. Functionally, this is expressed in the design of routines and data scoping. Object-oriented designs have extended the applicability of this concept into class definitions that give a collection of data bundled together with the operations that can manipulate it. The access that outside functions and objects have to the encapsulated data may vary; the intention is to hide data and restrict users to a narrow interface. Done effectively, the data and implementation of services can change drastically without the changes being visible to users. Several major changes to Kyacc's internal representations and algorithms were accomplished without major redesign of the class structures. Initially, LALR(1) states were generated directly; that is, every time a new state was generated, it was compared against the previous states for full equality in the set of items and, if true, the item sets were merged. This involved several unions of sets, which was inefficient. In a newer implementation of Kyacc, only LR(0) states are generated first and the look-ahead symbols for the items are computed using a version of an algorithm by Park, Choe and Chang [9]. We can confidently state that such a major change to Yacc would have been extremely difficult.

Secondly, the backend of Kyacc, the final phase of the parser generator in Figure 1, is so structured that it can be replaced by a different module to produce a different style of parser. Currently, it generates the same table-driven parsers that Yacc generates. It has also been tested to generate switch-statement style parsers. One interesting experiment would be to generate C++ based parsers. Johnson discusses the possibility of using C++ concepts to generate more flexible parsers associating arbitrary classes with grammar symbols [3].

Ryacc also went through major changes in algorithms and internal structures. Some of these changes were easier to accomplish than others. The first version of Ryacc did not account for the special nature of null reductions and was based on an algorithm that did not require readback machines. It became apparent that the algorithm could not truly parse ELALR(1) grammars, and so a major change had to be made. The hard part was to identify the problem. Modifying the code to include special reduction cases was straightforward because of the way the code was compartmentalized.

There were several versions of table packing and output during Ryacc's development. Some of the changes were made before actions became an elaborated hierarchy of the `lrEdge` class and some after. The initial changes were painful. We had to locate all the spots in the code where transitions were inspected and tweak the code. Often some spot would be missed, causing obscure bugs. Once the hierarchy was established and virtual functions used for the transition processing, it became easier to change behavior by changing those functions.

## 4. Summary

In reality, Meyer's four-step approach to object-oriented design is an iterative process, both within one step and between the steps [7]. The iteration is especially noticeable in redesigns of existing applications. A redesign is the result of pondering the problem, studying the existing implementation, applying past experience, and taking advantage of insights and experimentation. The clarification of the role of some objects can lead to discoveries that give rise to new objects or changes in the design of existing objects, in our experience with parser generators. Thus the final design is built from initial thin scaffolding to a final solid structure through iterative refinement and correction. The effort spent in iterating over that design process paid off later during the implementation. The class structures and their relationships rarely came in our way of making enhancements or replacing some basic algorithms with newer ones.



Smalltalk was useful for rapidly building an initial design of the class structures and their relationships in Kyacc. Later, the implementation was done in C++ to provide for the flexibility and efficiency needed in several stages of parser generators. Our implementations of parser generators in C++ used only types and encapsulation provided by C++ classes, single inheritance, overloaded operators, inline functions, macro expansions for generics, libraries in C++ 2.0, static class members, and statically initialized objects. Multiple inheritance was not used. Prototypes of Kyacc and Ryacc are available internally at the moment and they meet the stated objectives for redesigning and implementing Yacc. It is too early to tell, however, how successful they will be as production quality parser generators that employ the latest algorithms and address the current needs.

## References

1. Burke, M.G. and Fisher, G.A. A practical method for syntactic error diagnosis and recovery. *ACM SIGPLAN Notices* 17, 6 (June 1982), 67-78.
2. Goldberg, A. and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass, 1983.
3. Johnson, S.C. Yacc Meets C++. *Computing Systems* 1, 2 (1988), 159-167.
4. Johnson, S.C. Yacc—yet another compiler compiler. Computing Science Technical Report No. 32, AT&T Bell Laboratories, Murray Hill, N. J., 1975.
5. Kernighan, B.W. and Ritchie, D.M. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, N. J, 1978.
6. LaLonde, W.R. Constructing LR parsers for regular right part grammars. *Acta Informatica* 11 (1979), 177-193.
7. Meyer, B. *Object-Oriented Software Construction*. Prentice-Hall International, Englewood Cliffs, N.J., 1988.
8. Nakata, I. and Sassa, M. A simple realization of LR-parsers for regular right part grammars. *Information Processing Letters* 24 (1987), 113-120.
9. Park, J.C.H., Choe, K.M., and Chang, C.H. A new analysis of LALR formalisms. *ACM Trans. Programming Languages and Systems* 7, 1 (1985), 159-175.
10. Parnas, D.L. On the criteria to be used in decomposing systems into modules. *Comm. ACM* 15, 12 (1972), 1053-1058.
11. Sethi, R. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, Mass., 1989.
12. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, Mass, 1986.
13. Wirth, N. The module: a system structuring facility in high-level programming languages. In *Language Design and Programming Methodology*, Lecture Notes in Computer Science 79, Springer-Verlag, New York, 1979, pp. 1-24.

The first part of the paper describes the motivation for the work. The second part describes the system architecture. The third part describes the implementation. The fourth part describes the evaluation. The fifth part describes the conclusions.

# References

1. [1] ...
2. [2] ...
3. [3] ...
4. [4] ...
5. [5] ...
6. [6] ...
7. [7] ...
8. [8] ...
9. [9] ...
10. [10] ...
11. [11] ...
12. [12] ...
13. [13] ...
14. [14] ...
15. [15] ...
16. [16] ...
17. [17] ...
18. [18] ...
19. [19] ...
20. [20] ...

# GPERF: A Perfect Hash Function Generator

Douglas C. Schmidt

Department of Information and Computer Science

University of California, Irvine \*

Irvine, CA 92717

(schmidt@ics.uci.edu)

## Abstract

`gperf` is a perfect hash function generator written in C++. It translates an  $n$  element user-specified keyword set  $W$  into a perfect hash function  $F$ .  $F$  uniquely maps keywords in  $W$  onto the range  $0..k-1$ , where  $k \geq n$ . If  $k = n$  then  $F$  is a *minimal* perfect hash function. `gperf` generates a  $k$  element static lookup table and either a pair of C functions or a C++ class. The generated code determines whether any particular string  $s$  occurs in  $W$ , using at most one probe into the lookup table.

`gperf` currently generates the reserved keyword recognizer for lexical analyzers in several production and research compilers and language processing tools, including GNU C, GNU C++, GNU Pascal, GNU Modula 3, and GNU indent.

This paper provides an overview of perfect hashing and discusses the impact of C++ on the overall program design and maintenance effort. It also describes the interface, features, and implementation strategies incorporated in `gperf` and presents the results from an empirical comparison between `gperf`-generated recognizers and several other typical reserved word lookup techniques.

C++ source code for `gperf` is available via anonymous ftp from `ics.uci.edu` (128.195.1.1). `gperf` is also distributed along with the GNU `libg++` library. A highly portable, functionally equivalent K&R C version of `gperf` is also archived in `comp.sources.unix`, volume 20.

## 1 Static Search Structures

A *static search structure* is an Abstract Data Type (ADT) with certain fundamental operations, *e.g.*, *initialize*, *insert*, and *retrieve*. Conceptually, all insertions occur before any retrievals.<sup>1</sup> It is a useful data structure for representing *static search sets*. Static search sets occur frequently in system software applications. Typical static search sets include compiler reserved words, assembler instruction mnemonics, and built-in shell interpreter commands. Search set members, called *keywords*, are only inserted into the structure once and are not generally modified at run-time.

Numerous static search structure implementations exist, *e.g.*, arrays, linked lists, binary search trees, digital search tries, deterministic finite-state automata and hash tables [Knu73]. Different approaches offer trade-offs between space utilization and search time efficiency. For example, an  $n$  element sorted array is space efficient, though the average-case time complexity for retrieval operations using binary search is proportional to  $\log n$ . Conversely, chained hash table implementations

\*This work was supported in part by the National Science Foundation under grant CCR-8704311 with cooperation from the Defense Advanced Research Projects Agency under Arpa order 6108, program code 7T10; National Aeronautics and Space Administration under grant NSG-5123; National Science Foundation under grant DCR-8521398; University of California under the MICRO program; Hughes Aircraft; and TRW.

<sup>1</sup>In practice, `gperf` generates a static array containing search set keywords and any associated attributes specified by the user. Thus, there is essentially no execution-time cost for the insertions.

often locate a table entry in constant time, but typically impose additional memory overhead and exhibit poor worst case performance [AHU74, pp. 111–114].

*Minimal perfect hash functions* provide a theoretically optimal solution for a particular class of static search sets. A minimal perfect hash function is defined by two properties:

- It allows keyword recognition in a static search set using at most one probe into the hash table. This represents the “perfect” property.
- The actual memory allocated to store the keywords is precisely large enough for the keyword set and *no larger*. This is the “minimal” property.

For most applications it is substantially easier to generate *perfect* hash functions than *minimal perfect* hash functions [CO82]. Moreover, *non-minimal* perfect hash functions frequently execute faster than minimal ones in practice [Cic80a, Cic80b]. This phenomenon occurs since searching a sparse keyword lookup table increases the probability of locating a “null” entry, thereby reducing string comparisons. **gperf**’s default behavior generates *near-minimal* perfect hash functions for keyword sets. However, **gperf** provides many options that allow users to control the degree of minimality and perfection.

Static search sets often exhibit relative stability over time. For example, Ada’s 63 reserved words have remained constant for nearly a decade. It is frequently worthwhile to expend concerted effort building an optimal search structure *once*, if it subsequently receives heavy use *multiple* times. **gperf** removes the drudgery associated with constructing time and space efficient keyword recognizers by hand. It has proven a practical software tool and **gperf**-generated recognizers currently appear in several production and research compilers, including GNU C, GNU C++, GNU Pascal, and GNU Modula 3 [Tie89, Sta89].<sup>2</sup> Each compiler utilizes **gperf** to automatically generate a static search structure that efficiently identifies their respective reserved keywords. Other known **gperf** applications include:

- **gperf** generates a perfect hash function for 15,400 “Medical Subject Headings” used to index journal article citations in MEDLINE, a large bibliographic database of the biomedical literature maintained by the National Library of Medicine. Generating this perfect hash function takes approximately 16 minutes of wallclock time on a Sun 4/260.
- **IRC**, an International Real-time Conference system operating over the Internet.
- **gperf**-generated hash functions are available for assembly mnemonics in the 80x86, 680x0, Z8000, and MIPS RISC instruction sets.
- The GNU **indent** C code reformatting program, where the inclusion of perfect hashing sped up the program by an average of 10 percent. The GNU **pic** program also uses a **gperf** generated recognizer.
- A public domain program converting double precision FORTRAN to/from single precision uses **gperf** to handle subroutine names that depend on the types of their arguments, *e.g.* **sgefa** versus **dgefa** in the LINPACK benchmark. Each token corresponding to a subroutine name is recognized via **gperf** and replaced with the version for the other precision.

## 2 Interacting with gperf

**gperf** reads a set of keywords and optional attributes from a *keyfile* or from the standard input. Keywords are specified as character strings; keyword attributes can be any C literals. For example, keywords in Figure 1 below represent months of the year. Associated attributes in this figure include the number of leap year and non-leap year days in each month, as well as the months’ ordinal numbers, *i.e.*, january = 1, february = 2, ..., december = 12.

<sup>2</sup>The latter two compilers are not yet part of the official GNU distribution.



```
%{
/* The following code is inserted directly into the output. */
#include <string.h>
/* GPERF command-line options: -L C++ -p -a -n -t -o -j 1 -k 2,3 -N Lookup_Month */
}%
struct months { char *name; int number; int days; int leap_days; };
%%
january,      1,      31,      31
february,     2,      28,      29
march,        3,      31,      31
april,        4,      30,      30
may,          5,      31,      31
june,         6,      30,      30
july,         7,      31,      31
august,       8,      31,      31
september,    9,      30,      30
october,     10,      31,      31
november,    11,      30,      30
december,    12,      31,      31
```

Figure 1: Sample `gperf` Input File.

`gperf` attempts to create a perfect hash function that recognizes keywords with at most a single probe into the lookup table. If `gperf` succeeds it generates a C++ class with member functions that calculate hash values and perform table lookup. Various command-line options modify `gperf`'s default input and output format, *e.g.*, it can also produce K&R or ANSI C code.<sup>3</sup>

By default, `gperf` generates a lookup table scheme implemented using an array of keywords. This produces fast code, with less emphasis on efficient space utilization. However, several options permit trade-offs that reduce storage space and/or execution-time. For example, expanding the generated table size produces a larger, sparser lookup table, generally yielding faster keyword searches. Conversely, `gperf` can also generate a `switch` statement that potentially reduces data space storage size. Depending on the underlying compiler's optimization capabilities, using the `switch` statement may actually produce smaller and faster code compared with the array-based lookup table method.

In addition, `gperf` handles duplicate keyword entries and unresolved keyword collisions by generalizing the `switch` statement scheme. Support for duplicate entries is particularly useful when keywords are distinguishable only via secondary key comparisons. `gperf` treats duplicate keywords as equivalence classes and generates code containing a `switch` statement with cascading `if-else` comparisons for non-unique `case` labels. The user can then edit the generated code by hand or via an automated script to completely disambiguate the keywords.

## 2.1 Input and Output Format

`gperf`'s input format is structurally similar to the UNIX utilities `lex` and `yacc`:

```
declarations and text inclusions
%%
keywords and optional attributes
%%
auxiliary functions
```

<sup>3</sup>It is beyond the scope of this paper to describe all of `gperf`'s many options. The `gperf` user's manual supplied along with the distribution provides additional information.

A pair of consecutive `%%` symbols separate declarations from the list of keywords and attributes. C or C++ source and comments are included into the generated output file by enclosing the text inside `%{ %}` pairs. An optional user-supplied `struct` declaration is placed at the end of the declaration section to enable typed attribute initialization (see Figure 1).

It is possible to omit the declaration section entirely. In this case the keyfile starts with the first keyword line. This format style is useful for building keyword set recognizers that possess no additional attributes. For example, a perfect hash function for “frequently occurring English words” can efficiently filter out uninformative words from consideration in a “key-word-in-context” indexing application [Knu73, p. 437].

Lines containing keywords and associated attributes appear in the second keyfile format section. The first field of each line always contains the keyword itself, any additional attribute fields follow up to the end-of-line marker. Field separators are user-specified, defaulting to the comma and newline characters. Attribute fields initialize components of the user-defined `struct` provided in the declaration section.

All text in the optional third section is included verbatim into the generated output file, starting immediately after the final `%%` and extending to the end of the input file. Naturally, it is the user’s responsibility to insure that the inserted code is valid C or C++.

Figure 1 presents an example keyfile containing months of the year and their attributes. Appendix A shows the generated C++ code output by `gperf`. More substantial examples are available with the `gperf` distribution, where keyfiles for Ada, C, Pascal, C++, Modula 2, and Modula 3 reserved keywords are included in the release.

### 3 C++ Influence

`gperf` was initially inspired by Keith Bostic’s C program “perfect” distributed to `net.sources` in 1984. Early versions of `gperf` were also written in C; they provided additional options and a more flexible user interface. The current version is a completely revised, enhanced, and extended implementation written in approximately 3,400 lines of C++ [Str86]. The C++ rewrite occurred as `gperf` grew in size and complexity. Over time, the original C program became increasingly unstructured and assuring future bug-free development necessitated redesign of the internal data structures, algorithms, and overall program decomposition. C++ offered the necessary data abstraction facilities to support a major revision.

`gperf` is continually updated and improved, often reflecting changes to C++ itself. For example, the current version employs `cfront` 2.0 features such as *multiple inheritance*, which clarifies the overall program structure and reduces the need for global objects, *static member functions*, which further remove the need for global variables and reduce function-call overhead, and *overloaded operator new*, which defines an efficient buffered memory allocation cache that minimizes the more expensive calls to `malloc`.

#### 3.1 Reusable Class Componentry

Figure 2 illustrates `gperf`’s current multiple inheritance hierarchy. `gperf` is constructed from reusable components that also serve as base-classes in a “forest”-style library [Lea88]. Each of these classes evolved “bottom-up” from special-purpose utilities into reusable software components. Recycling, revising, and refining useful abstractions over time is a simple and honorable way for library designers and application programmers to build reusable component collections. Particularly noteworthy general-purpose classes shown in Figure 2 include the following ADTs:

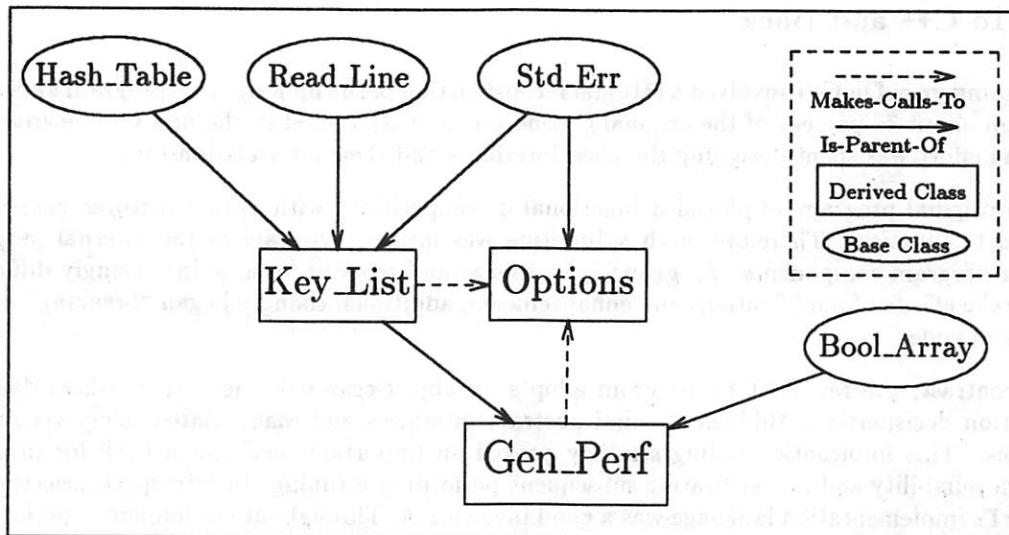


Figure 2: Inheritance Hierarchy in gperf

**Hash\_Table:** provides a search structure implemented via double hashing [Knu73, pp. 521–526]. During program initialization **gperf** uses this class to detect keyfile entries that are guaranteed to produce the same hash values. These “static” duplicates occur whenever keywords possess identical sets of characters used to compute their hash values. Unless the ‘-D’ (handle duplicate keywords) option is enabled, attempting to generate a perfect hash function for keywords with non-unique hash values is an exercise in futility!

**Read\_Line:** efficiently reads an arbitrarily long line from an input file. Each line in the keyfile contains a single keyword followed by any optional attributes. A pointer to a dynamically allocated buffer containing the line’s contents is returned. A recursive auxiliary member function insures only one call to **new** is made per input line, *i.e.*, there is no need for reallocating and resizing buffers dynamically. This routine proved useful enough to be incorporated as an extension in the GNU libg++ **stream** library [Lea89].

**Std\_Err:** standardizes the formatting of error messages throughout **gperf**. This class generalizes the functionality of the UNIX **perror** library routine. Static class member function **void Std\_Err::report\_error(const char \*, ...)** interprets a **printf**-style format string containing directives that provide a uniform error handling facility. Standard services include: (1) aborting the program, (2) calling function-pointers passed as parameters, (3) displaying the name used to invoke the main program, (4) printing appropriate system error messages corresponding to failed system and library calls, and (5) outputting various common data types passed as arguments to the variadic function.

**Bool\_Array:** employs an efficient *Iteration Numbering* technique to dynamically identify keywords possessing duplicate hash values during program execution. On large input keyfiles, **gperf** spends approximately 90 to 99 percent of its time checking whether or not changes to certain internal data structures eliminate duplicate hash values. It is therefore important that this boolean array ADT exhibit minimal overhead. The evolution of this class illustrates an important tenet of object-oriented design: “first determine a clean set of operations and interfaces, then successively tune the implementation.” This policy of optimizing performance without compromising program clarity sped **gperf** up by an average of 25 percent; the general technique is described in Section 4.2 below.

## 3.2 To C++ and Back

Redesigning **gperf** in C++ involved a rational reconstruction of the underlying C program's structure. Although about 75 percent of the original C code was directly reused in the first C++ rewrite, most time and effort was spent designing the class interfaces and their interrelationships.

The original program employed a functional decomposition, with data structures passed from function to function. Therefore, each subroutine was intimately aware of the internal properties of other program components. As **gperf** grew in size and scope it became increasingly difficult to localize the effects of modifications and enhancements; additional changes began "breaking" existing sections of code.

In contrast, the revised C++ program adopts an object-centered orientation, where data representation decisions are "hidden" behind abstract interfaces and manipulated solely via member functions. This information hiding strategy proved an important mechanism both for increasing program reliability and for facilitating subsequent performance tuning. In retrospect, selecting C++ as **gperf**'s implementation language was a good investment. Throughout the numerous performance enhancements and feature additions the public member functions for major class components have remained relatively stable, greatly reducing interface errors between modules.

**gperf**'s evolution has come full circle. Since C++ is not immediately available on all platforms and systems, a highly portable K&R C version of **gperf** was written.<sup>4</sup> Called **cperf**, this new C program is developing in parallel with **gperf**, directly reusing large sections of C++ code.

**cperf** was developed from the same design decomposition used in the C++ version. One might argue that **gperf**'s reusable component design could have emerged directly from the Keith Bostic's original C program. In other words, its design should have been independent from its implementation language. In practice, it became far more natural to conceptualize and realize the program decomposition in an intellectually manageable fashion using C++, since C++ actively *supports* data abstraction and promotes effective Abstract Data Type encapsulation techniques [Str88, pp. 10-11][AHU83].

The **cperf** rewrite exhibits almost 100 percent code reuse, with major cosmetic changes occurring in program interfaces, *e.g.*, substituting function prototypes with old-style K&R C function declarations, modifying classes and member functions to become structs and regular C functions, and replacing inheritance relations with stand-alone objects. Needless to say, it was psychologically painful to *remove* levels of abstraction when migrating the code from C++ back to C!

## 4 Implementation Issues

Few articles describe perfect hash function generator design and implementation in detail [CO82, Sag86, Cic80a]. Even fewer generators are widely available in source code form. This section examines **gperf**'s internal data structures, outlines its perfect hash function generation algorithm, and discusses the program's current limitations.

### 4.1 Internal Data Structures

Understanding **gperf**'s implementation requires a description of two important internal constructs: the *associated values* array and individual keyword *signatures*. Every user-specified keyword and its attributes are stored as a node in a linked list. However, **gperf** only considers a subset of each keyword while searching for a solution. Referred to as the keyword's *keysig*, this subset is created

<sup>4</sup>This version is now available from the comp.sources.unix archives, volume 20.



and stored in each linked list node when the keyfile is initially processed. The keysig is actually a *multiset* or *bag*, as it may contain multiple occurrences of certain characters. A keysig represents the subset of a keyword's characters used to compute its hash value by the automatically generated recognition function.

A related data structure is the *associated values* array, indexed by keysig characters. This array is declared as "unsigned int asso\_values[MAX\_ASCII\_SIZE]," constructed internally by `gperf`, referenced frequently during program execution, and later output as a static array local to the generated hash function. `gperf` repeatedly reassigns different values to certain `asso_values` elements specified by characters in keyword keysigs when searching for a perfect hash function solution. At every step in the algorithm the contents in the `asso_values` array represent the current associated values' configuration.

A perfect hash function is produced when `gperf` determines an associated values configuration that maps each keyword's keysig to a unique location within the generated lookup table. The resulting perfect hash function returns an unsigned integer value in the range  $0..k$ , where  $k$  is the maximum keyword hash value. In general, the generated lookup table's *load factor* increases as  $k$  approaches  $n$  from above.<sup>5</sup> When  $k = n$  the table is a *minimal* perfect hash function.

Users may explicitly specify which keyword index positions are used for the generated hash function, thereby determining the hash function by specifying the keysig elements. `gperf` then searches for an associated values configuration that results in a unique mapping from keywords onto hash values.

By default, hash values are computed by combining a keyword's length with the associated values of its keysig, *i.e.*, the hash function adds the associated value of a keyword's first index position plus the associated value of its last index position to its length:

```
hash_value = asso_values[keyword[0]]
            + asso_values[keyword[length - 1]] + length;
```

Other combinations are often necessary in practice, however [JO80]. For example, a conflict arises between the C++ reserved words `delete` and `double`, requiring the use of:

```
hash_value = asso_values[keyword[0]] + asso_values[keyword[1]]
            + asso_values[keyword[length - 1]] + length;
```

to generate a perfect hash function for C++ keywords. Choosing different key positions is readily expressed in `gperf`'s command-line syntax, *e.g.*, appropriate keysigs for C++ keywords are specified by the '-k 1,2,\$' option.

Users can also instruct `gperf` to consider *all* character positions when computing each keyword's hash value. Keywords shorter than certain specified index positions are properly handled, since character positions exceeding the keyword's length are simply ignored in the generated hash function.

## 4.2 Main Algorithm

`gperf`'s three main phases for generating perfect hash functions are:

1. Process command-line options, read keywords and attributes, and initialize internal data structures.
2. Perform a non-backtracking, heuristically guided search for a perfect hash function.
3. Generate formatted C or C++ code according to user-specified options.

---

<sup>5</sup>The load factor is represented by the ratio  $\frac{\text{number of keywords}}{\text{total table size}}$ .

This section describes **gperf**'s non-backtracking search algorithm in detail.

**gperf** iterates sequentially through the list of  $1 \leq i \leq n$  keywords, where  $n$  equals the total number of keywords. During each iteration **gperf** attempts to extend the set of uniquely mapped keywords by 1. It succeeds if the hash value computed for keyword  $i$  does not conflict with the previous  $i - 1$  uniquely hashed keywords, i.e.:

```
for  $i \leftarrow 1$  to total keywords loop
  if hash-value( $i^{\text{th}}$  key) conflicts with hash-value( $1^{\text{st}}$  key) thru hash-value( $(i - 1)^{\text{th}}$  key) then
    try changing certain associated values to resolve conflict
  end if
end loop
```

The algorithm terminates and generates a perfect hash function when  $i = n$  and no unresolved hash collisions remain. The *best-case* asymptotic time-complexity for this algorithm is  $\Omega(n)$ , i.e., linear in the number of keywords.

However, nearly all actual keyword sets produce hash conflicts during the search process. As illustrated in the algorithm outlined above, **gperf** attempts to resolve collisions between keywords by modifying certain associated values. **gperf** is selective when changing associated values. First, it only considers characters comprising the *disjoint union* of the conflicting keywords' keysigs<sup>6</sup> (note that no other associated values can possibly resolve the collision at this point). Second, as a heuristic, the disjoint union is sorted by increasing frequency of occurrence, so that least-used characters are changed before more frequently used characters.

A perfect hash function is achieved if systematic changes to the associated values' configuration eliminates all keyword collisions upon reaching the end of the keyword list. The *worst-case* asymptotic time-complexity for this algorithm is  $O(n^3l)$ , where  $l$  is the size of the largest disjoint union between conflicting keyword keysigs. Such worst-case behavior rarely occurs in practice.

Perfect hash function algorithms are typically sensitive to the order that keywords are considered [Cic80b, CO82]. **gperf** mitigates this effect by optionally reordering the keywords. This reordering pre-pass applies two heuristics that potentially prune the search space by handling inevitable collisions early in the generation process. As a result, **gperf** runs faster on many keyword sets and the perfect hash function range usually decreases. On the other hand, if the number of keywords is large this reordering may actually *increase* **gperf**'s execution time, since collisions begin earlier and continue throughout the remainder of keyword processing.

The two-stage reordering process first sorts the keyword list by decreasing frequency of keysig characters' occurrence. A second reordering pass then places keys with "already determined keysig values" earlier in the keylist. Cichelli provides additional details and rationalizations for this procedure in [Cic80b, p. 18].

### 4.3 Iteration Numbering

When processing a large keyfile that evokes many collisions **gperf** spends about 90 to 99 percent of its time in one routine. This routine determines how changes to associated values affect previously hashed keywords. **gperf** employs a novel boolean array ADT called **Bool\_Array** to expedite this process. The C++ interface for the **Bool\_Array** class is depicted in Figure 3. Only one copy of **Bool\_Array** is required, so all class data and member functions are declared with storage class **static**. This reduces run-time overhead since no "this" pointer is passed during function calls.

Class member function **Bool\_Array::in\_set(int)** efficiently detects duplicate keyword hash values for a given associated values configuration, returning non-zero if a value is already in the set

<sup>6</sup>The disjoint union of two keysigs  $\{A\}$  and  $\{B\}$  is defined as  $\{A \cup B\} - \{A \cap B\}$ .

```

class Bool_Array
{
private:
    typedef unsigned short TYPE;    // Using unsigned short saves space
    static TYPE iteration_number;    // Current iteration count
    static TYPE *array;             // Dynamically allocated storage buffer
    static int size;                // Length of dynamically allocated array

public:
    Bool_Array(int k);              // Allocate a k element dynamic array
    ~Bool_Array(void);              // Returns dynamic memory to free store
    static int in_set(int value);    // Checks if 'value' is a duplicate
    static void reset(void);         // Reinitializes all set elements to FALSE
};

```

Figure 3: Boolean Array ADT.

and zero otherwise. Whenever a duplicate is detected the boolean array must be “reset” for the ensuing iteration. If many hash conflicts occur `Bool_Array::reset()` is executed frequently during the duplicate detection and elimination process.

Processing large keyword sets tends to require a maximum hash value  $k$  that is often *much* larger than  $n$ , the total number of keywords. Due to the large range, it becomes expensive to explicitly reset all elements in `Bool_Array::array`, especially when the number of keywords actually checked for duplicate hash values is comparatively small. Class `Bool_Array` is implemented to avoid explicitly reinitializing the entire array via a technique called *Iteration Numbering*. The technique, which is also useful as a strategy for repeated depth-first searches of graph structures [AHU83], operates as follows:

1. The class constructor dynamically allocates space for  $k$  “unsigned short” integers and points `Bool_Array::array` at this memory. All  $k$  array elements in `Bool_Array::array` are initially assigned 0 (representing FALSE) and the `Bool_Array::iteration_number` counter is set to 1.
2. “Resetting” the ADT on subsequent iterations simply requires incrementing the value of `Bool_Array::iteration_number` by 1. The entire  $k$  array locations are only reinitialized to zero when the iteration number exceeds the range of an unsigned short integer, which occurs *very* infrequently in practice.
3. To check for duplicate keyword hash values the appropriate location in `Bool_Array::array` is referenced via the `Bool_Array::in_set(int)` member function. If the number stored at index position `Bool_Array::array[hash(keyword)]` is *not* equal to the current iteration number then that hash value is *not* already in the set. In this case, the current iteration number is immediately assigned to the array location specified by the hash value, thereby marking it as a duplicate if it is subsequently referenced during this particular iteration.
4. Otherwise, if the referenced value at location `Bool_Array::array[hash(keyword)]` is *equal* to the iteration number, a duplicate exists and the algorithm must try modifying a different associated value to resolve the conflict.

Implementing the Iteration Numbering scheme decreased `gperf`’s execution-time by an average of 25 percent for large keyfiles, compared with the previous method that explicitly “zeroed out” the entire boolean array’s contents on every reset.

#### 4.4 Current Compromises and Limitations

`gperf` is tuned for average-size keyfile input, *e.g.*, compiler reserved keywords. It runs extremely

Object File	Byte Count				
	text	data	bss	dynamic	total
control.o	88	0	0	0	88
binary.o	1,008	288	0	0	1,296
gperf.o	2,672	0	0	0	2,672
chash.o	1,608	304	8	1,704	3,624
patricia.o	3,936	0	0	2,272	6,208
comp-flex.o	7,920	56	16,440	0	24,416
trie.o	79,472	0	0	0	79,472
flex.o	3,264	98,104	16,440	0	117,808

Figure 4: Size of Object Files in Bytes

fast on small to medium size keyword sets up to approximately 1,000 keys, often requiring only a few CPU seconds to execute. In addition, internal data structure and algorithm enhancements now enable **gperf** to operate efficiently on *much* larger keyword sets, containing over 15,000 keywords.

Several other generators utilize some form of backtracking when searching for perfect or minimal perfect hash functions [CO82, Cic80a]. For example, Cichelli's minimal perfect hash function generation algorithm recursively attempts to find an associated values configuration that uniquely maps all  $n$  keywords to distinct integers in the range  $1..n$ . If computing the current keyword's hash value exceeds the table size constraint at any point during program execution the algorithm "backs up" to the previous keyword. It then proceeds by undoing selected hash table entries, re-assigning different associated values, and continuing to search for a solution. Unfortunately, even "intelligently-guided" exhaustive search quickly becomes impractical for large keyfiles. The exponential growth rate associated with the backtracking search process is simply too time consuming for non-trivial applications.

To enhance average-case performance and simplify the main algorithm **gperf** does not backtrack when keyword hash conflicts occur. Under certain circumstances, therefore, **gperf** processes the entire keyfile input *without* finding a unique hash mapping for every keyword, even if one exists. If a unique mapping is not found users have several choices. They can either run the program again, enabling different options in search of a perfect hash function, or they can *guarantee* a solution by instructing **gperf** to generate an *almost-perfect* hash function.

An almost-perfect hash function allows keywords to possess non-unique hash values. This technique is a compromise that trades increased *generated-C-code-execution-time* for decreased *perfect-hash-function-generation-time*.<sup>7</sup> Although the resulting hash function is no longer "perfect" it handles keyword membership queries efficiently, since only a small number of duplicates generally remain. A simple linear search is performed on the few duplicate keywords that hash to the same location. Linear search is appropriate since the vast majority of keywords still require at most one lookup table probe. Support for duplicate hash values is particularly useful for large input keyfiles or for sets whose keywords are very similar, *e.g.*, assembler instruction mnemonics. It permits **gperf** to operate on keyword sets that otherwise could not be handled.

## 5 Empirical Results

**gperf** facilitates rapid-prototyping by automating a common software operation (keyword recognition). However, a tool-generated recognizer is not necessarily advantageous for production-quality

<sup>7</sup> Almost-perfect hash functions also tend to produce much smaller lookup tables, which is important when main memory is scarce.



Executable Program	Input File									
	ET++.in		NIH.in		g++.in		idraw.in		cfront.in	
control.exe	38.8	1.00	15.4	1.00	15.2	1.00	8.9	1.00	5.7	1.00
trie.exe	59.1	1.52	23.8	1.54	23.8	1.56	13.7	1.53	8.6	1.50
flex.exe	60.5	1.55	23.9	1.55	23.9	1.57	13.8	1.55	8.9	1.56
gperf.exe	64.6	1.66	26.0	1.68	25.1	1.65	14.6	1.64	9.7	1.70
chash.exe	69.2	1.78	27.5	1.78	27.1	1.78	15.8	1.77	10.1	1.77
patricia.exe	71.7	1.84	28.9	1.87	27.8	1.82	16.3	1.83	10.8	1.89
binary.exe	72.5	1.86	29.3	1.90	28.5	1.87	16.4	1.84	10.8	1.89
comp-flex.exe	80.1	2.06	31.0	2.01	32.6	2.14	18.2	2.04	11.6	2.03
									9.2	2.04

Figure 5: CPU Time Required to Process Identifiers

applications unless the resulting executable code speed is competitive with typical alternative implementations. In fact, it has been argued that there are *no* circumstances where perfect hashing proves worthwhile [Keg86, p. 557]. To compare the efficacy of automatically generated perfect hash functions against other static search structure implementations 7 representative test programs were devised and executed on 6 large input files.

Each test program implemented the same function: a recognizer for the 71 GNU G++ reserved words. The function returns TRUE if a given input string is identified as a reserved word and FALSE otherwise. A simple C++ program measures and controls for I/O overhead:

```
int main (void)
{
    const int MAX_ID = 80; /* Larger than any identifier from the input. */
    char buf[MAX_ID];
    while (gets (buf))
        printf ("%s", buf);
}
```

The 7 other test programs are described below, listed by increasing order of average test input execution time. Figure 4 shows the size in bytes for each compiled object file.<sup>8</sup>

**trie.exe:** a program based upon an automatically generated table-driven trie created by my **trie-gen** utility included with the GNU libg++ distribution.

**flex.exe:** a flex-generated recognizer created with the **'-f'** (no table compaction) option, maximizing speed in the generated scanner.

**gperf.exe:** a gperf-generated recognizer created with the **'-a -D -S1'** options.<sup>9</sup>

**chash.exe:** a dynamic chained hash table lookup routine similar to the one that recognizes reserved words for **cfront 2.0**.

**patricia.exe:** a PATRICIA trie recognizer, where PATRICIA stands for "Practical Algorithm to Retrieve Information Coded in Alphanumeric" [Sed83, pp. 219-223]. My complete PATRICIA trie implementation is also available in the GNU libg++ class library distribution.

**binary.exe:** a carefully coded binary search routine that tries to minimize the number of complete string comparisons.

<sup>8</sup>Note that **patricia.o** and **chash.o** use dynamic memory, so the overall memory usage depends upon underlying representations of the freestore mechanism. These statistics are based upon the **malloc** routine from the GNU libg++ 1.37 library.

<sup>9</sup>These options mean "generate ANSI C prototypes (**'-a'**), handle duplicate keywords (**'-D'**), via a single switch statement (**'-S1'**)."

**comp-flex.exe:** a flex-generated recognizer created with the default `'-cem'` options, providing the highest degree of table compression.

All reserved word recognizer programs were tested on an otherwise idle Sun 4/260 with 32 megabytes of RAM; they were compiled by the GNU G++ 1.37 compiler with the `"-O -fstrength-reduce -finline-functions -fdelayed-branch"` options enabled.

Each input file contained a large number of words, both user-defined identifiers and keywords, organized with one word per line. The words were created by running the UNIX command `"tr -cs A-Za-z_ '\012'"` on the preprocessed source code cfront 2.0, GNU libg++ 1.37, the GNU G++ 1.37 compiler, the ET++ source code, the NIH class library, and the idraw utility source in the InterViews 2.6 distribution. The following test suite was produced:

Input File	Identifiers	Keywords	Total
ET++.in	624,156	350,466	974,622
NIH.in	209,488	181,919	391,407
g++.in	278,319	88,169	366,488
idraw.in	146,881	74,744	221,625
cfront.in	98,335	51,235	149,570
libg++.in	69,375	50,656	120,031

Figure 5 depicts the amount of time each implementation spent executing the test programs. The first number in each column represents the user-time CPU seconds for each recognizer; the second number is the ratio of user-time CPU seconds divided by the I/O control program timing. Note that the execution times for each technique are very consistent across the input test file suite.

Several tentative conclusions result from these empirical comparisons:

- The uncompact table-driven flex and trie implementations are both the fastest and the largest implementations, illustrating the time/space trade-off dichotomy. Applications where saving time is more important than conserving space may benefit from this approach.
- Despite having a load factor of only 0.45 percent the gperf-generated perfect hash function recognizer is both space and time efficient. While methods like tries and finite-automaton recognizers allow programmers to trade off space for time, perfect hashing does not overly penalize either. An empirical indication of this balance may be calculated from the data presented in the figures for the programs that did not allocate dynamic memory. The number of identifiers scanned per second per byte of executable program overhead was 5.6 for gperf, but less than 1.0 for trie, flex, and comp-flex.
- Since gperf generates a stand-alone recognizer function it is easily incorporated into an otherwise hand-coded lexical analyzer.
- Since automatic static search structure generators perform well in practice and are widely and freely available there seems little incentive to code keyword recognition functions by hand for most applications.

## 6 Future Directions

Fully automating the perfect hash function generation process remains the most significant unfinished extension to gperf. One approach is to replace gperf's current algorithm with a more exhaustive approach. Due to C++'s abstraction mechanisms this modification will not unduly disrupt the overall program structure. The perfect hash function generation module, `class GenPerf`, is essentially

independent from other program components, representing only about 10 percent of **gperf**'s overall source lines of code.

A more comprehensive, albeit computationally expensive, approach would utilize backtracking or enable alternative options and retry when failing to generate a perfect hash function. In practice most search sets are relatively small, so the utility of these proposed modifications remains an open question.

Another potentially worthwhile feature involves having **gperf** determine which keyword index positions to select in order to generate the most time or space efficient hash functions. Currently, the user must explicitly select these positions via command-line arguments. **gperf**'s output routines should be extended to generate code for other languages, *e.g.*, an Ada package or Modula 2 module.

## Acknowledgments

In addition to Keith Bostic, who initially inspired **gperf**, special thanks is extended to Michael Tiemann and Doug Lea. Michael wrote the GNU G++ compiler and Doug gave me a forum in GNU libg++ to exhibit my creation and he also commented on drafts of this paper. Adam de Boor and Nels Olson also contributed many insights that greatly helped improve the quality and functionality of **gperf**. Finally, Vern Paxson provided an efficient **flex** input specification file for the GNU C++ keywords.

## References

- [AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design of Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.
- [Cha86] C.C. Chang. A Scheme for Constructing Ordered Minimal Perfect Hashing Functions. *Information Sciences*, 39:187-195, 1986.
- [CHK85] G.V. Cormack, R.N.S. Horspool, and M. Kaiserwerth. Practical Perfect Hashing. *Computer Journal*, 28:54-58, January 1985.
- [Cic80a] Richard J. Cichelli. Author's Response to "On Cichelli's Minimal Perfect Hash Functions Method". *Communications of the ACM*, 23:729, December 1980.
- [Cic80b] Richard J. Cichelli. Minimal Perfect Hash Functions Made Simple. *Communications of the ACM*, 23:17-19, January 1980.
- [CO82] C.R. Cook and R.R. Oldehoeft. A Letter Oriented Minimal Perfect Hashing Function. *SIGPLAN Notices*, 17:18-27, September 1982.
- [DKM<sup>+</sup>] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R.E. Tarjan. Dynamic Perfect Hashing: Upper and Lower Bounds. In *29<sup>th</sup> Focus on Computer Science*, pages 524-531.
- [Jae81] G. Jaeschke. Reciprocal Hashing: A Method for Generating Minimal Perfect Hashing Functions. *Communications of the ACM*, 24:829-833, December 1981.
- [JO80] G. Jaeschke and G. Osterburg. On Cichelli's Minimal Perfect Hash Functions Method. *Communications of the ACM*, 22:728-729, December 1980.

- [Keg86] Jeffrey Kegler. A Polynomial Time Generator for Minimal Perfect Hash Functions. *Communications of the ACM*, 29:556-557, June 1986.
- [Knu73] Donald Knuth. *The Art of Computer Programming, VOL 3: Sorting and Searching*. Addison-Wesley, 1973.
- [Lea88] Doug Lea. libg++, The GNU C++ Library. In *USENIX C++ Conference Proceedings*, pages 243-256. USENIX Association, October 1988.
- [Lea89] Doug Lea. User's Guide to GNU C++ Class Library. *Free Software Foundation*, 1989.
- [Sag86] Thomas J. Sager. A Polynomial Time Generator for Minimal Perfect Hash Functions. *Communications of the ACM*, 28:523-532, December 1986.
- [Sed83] Robert Sedgewick. *Algorithms*. Addison-Wesley, 1983.
- [Spr77] R. Sprugnoli. Perfect Hashing Functions: A Single Probe Retrieving Method for Static Sets. *Communications of the ACM*, 11:841-850, November 1977.
- [ST85] R.W. Sebesta and M.A. Taylor. Minimal Perfect Hash Functions for Reserved Word Lists. *SIGPLAN Notices*, 20:47-53, September 1985.
- [Sta89] Richard M. Stallman. Using and Porting GNU CC. *Free Software Foundation*, 1989.
- [Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.
- [Str88] Bjarne Stroustrup. What is Object-Oriented Programming? *IEEE Software*, pages 10-20, May 1988.
- [Tie89] Michael D. Tiemann. User's Guide to GNU C++. *Free Software Foundation*, 1989.

## A Sample gperf Output

Figure 6 below depicts the gperf-generated minimal perfect hash function corresponding to the input file depicted in Figure 1. Execution time was negligible on a Sun 4/260, i.e., 0.0 user and 0.0 system time.



```

/* C++ code produced by gperf version 2.3 (GNU C++ version) */
/* Command-line: gperf -L C++ -p -a -n -t -o -j 1 -k 2,3 -W Lookup_Month months.gperf */
/* The following code is inserted directly into the output. */
#include <string.h>
/* GPERF command-line options: -L C++ -p -a -n -t -o -j 1 -k 2,3 -W Lookup_Month */
struct months { char *name; int number; int days; int leap_days; };
/* maximum key range = 12, duplicates = 0 */

class Lookup_Month
{
private:
    const int MIN_WORD_LENGTH, MAX_WORD_LENGTH, MIN_HASH_VALUE, MAX_HASH_VALUE;
    unsigned int hash (const char *str, int len);
public:
    Lookup_Month (void):
        MIN_WORD_LENGTH (3), MAX_WORD_LENGTH (9), MIN_HASH_VALUE (0), MAX_HASH_VALUE (11) {}
    struct months *operator () (const char *str, int len);
};

unsigned int
Lookup_Month::hash (register const char *str, register int len)
{
    static unsigned char asso_value[] =
    {
        12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
        12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
        12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
        12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
        12, 2, 9, 5, 12, 0, 12, 7, 12, 12, 12, 12, 6, 12, 1, 11, 0, 12, 2, 12, 5, 0, 0, 12,
        12, 6, 12, 12, 12, 12, 12, 12,
    };
    return asso_value[str[2]] + asso_value[str[1]];
}

struct months *
Lookup_Month::operator () (register const char *str, register int len)
{
    static struct months wordlist[] =
    {
        {"september", 9, 30, 30},
        {"june", 6, 30, 30},
        {"april", 4, 30, 30},
        {"january", 1, 31, 31},
        {"march", 3, 31, 31},
        {"december", 12, 31, 31},
        {"july", 7, 31, 31},
        {"august", 8, 31, 31},
        {"may", 5, 31, 31},
        {"february", 2, 28, 29},
        {"october", 10, 31, 31},
        {"november", 11, 30, 30},
    };
    if (len <= MAX_WORD_LENGTH && len >= MIN_WORD_LENGTH)
    {
        register int key = hash (str, len) - MIN_HASH_VALUE;

        if (key <= (MAX_HASH_VALUE - MIN_HASH_VALUE) && key >= 0)
        {
            register char *s = wordlist[key].name;

            if (*s == *str && !strcmp (str + 1, s + 1))
                return &wordlist[key];
        }
    }
    return 0;
}

```

Figure 6: Minimal Perfect Hash Function Generated by gperf

The first part of the paper discusses the importance of the C++ language in the context of the 1990 USENIX C++ Conference. It highlights the language's role in system programming and its growing adoption in the industry.

The second part of the paper presents a detailed analysis of the C++ standard library. It examines the various components of the library, including the standard containers, the standard algorithms, and the standard input/output streams. The analysis focuses on the efficiency and portability of these components.

The third part of the paper describes the implementation of the C++ standard library. It details the design decisions and the implementation techniques used to create a high-performance and portable library. The implementation is based on the C++ standard and is designed to be compatible with other C++ implementations.

The fourth part of the paper presents the results of the implementation. It shows the performance of the library on various benchmarks and compares it to other C++ implementations. The results demonstrate the efficiency and portability of the implementation.

The fifth part of the paper discusses the future of the C++ language. It explores the ongoing work on the C++ standard and the challenges facing the language. It also discusses the potential for new C++ features and the impact of these features on the language.

The sixth part of the paper concludes the paper and summarizes the main findings. It emphasizes the importance of the C++ language and the need for continued research and development in the area of system programming.

# C++ and Operating Systems Performance: A Case Study\*

Vincent F. Russo, Peter W. Madany, and Roy H. Campbell  
University of Illinois at Urbana-Champaign  
Department of Computer Science  
1304 W. Springfield Avenue, Urbana, IL 61801

## Abstract

Object-oriented design and programming has many software engineering advantages. Its application to large systems, however, has previously been constrained by performance concerns. The *Choices* operating system, which has over 75,000 lines of code, is object-oriented and programmed in C++. This paper is a case study of the performance of *Choices*.

## 1 Introduction

Proponents of object-oriented design and programming proclaim advantages including design and code reuse and rapid prototyping. Others hesitate to adopt the approach because of the cost of retraining developers, rumors of poor performance, and the lack of adequate tools for building large, efficient object-oriented systems. The C++ language[14] has been widely promoted as an object-oriented language with minimal overhead. In 1987, members of the *Choices*[4] research team set out to build an object-oriented multiprocessor operating system written in C++. This work represents both an attempt to validate claims made by proponents of object-oriented programming and an attempt to investigate object-oriented operating systems. The results of this ongoing experiment include the *Choices* operating system, which contains about 75,000 lines of C++ code written by a group of approximately 15 graduate students.

In this paper we will measure and analyze both the C++ source code and the performance of *Choices*. We will then compare both to a UNIX<sup>TM</sup> [2] system which provides similar features and performs similar tasks on the same hardware. The conclusion will show how the evidence from this research supports the claim that, despite the lack of tools for such work, it is possible to build efficient, high-performance object-oriented software in C++ and that such software can compete with commercial software.

## 2 Overview of *Choices*

*Choices* is a complete operating system, which runs stand-alone on a particular computer without depending on any proprietary software. Designed with object-oriented

---

\*This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG1471 and NAG 1-163.

techniques[3] and built using an object-oriented programming language, *Choices* provides a transparent object-oriented application interface to protected system resources. This interface allows dynamic method invocations on the collection of objects which constitutes the kernel of the system.

*Choices* has been designed as a framework that supports experimentation with modern operating system technology. In particular, *Choices* is multi-threaded for efficient multi-processor use[11]. It offers low-level support for light-weight context switching from an application process to another that shares the same address space, to a system process, or to an interrupt process. The virtual memory system[12] uses machine-independent page tables, shared paged "segments" that may be simultaneously located in multiple virtual address spaces, and scatter-gather I/O for page fault handling. Each segment can have its own backing storage which may reside on various devices including a disk, a disk partition, a file, or physical memory. Distributed virtual memory[5] supports coherent sharing of data between applications over networks like Ethernet. The file system class hierarchy[7, 8] includes support for disks, partitions, and files and directories conforming to the System V UNIX[15], BSD 4.2 UNIX[9], or MS-DOS<sup>TM</sup> [10] standards. Programs can use read and write operations to access the files or files can be memory-mapped. Currently, *Choices* has networking support for Ethernet, UDP/IP, and TCP/IP. For users who want to run many existing programs, *Choices* provides a UNIX compatibility library. An X11<sup>TM</sup> windowing facility is planned.

The *Choices* design has been influenced by, but is not a reimplement of, systems like UNIX[2] and Mach[1]. Instead, the goal of *Choices* is to further parallel high-performance multiprocessor research by exploiting the customizability of object-oriented systems.

*Choices* already runs on Encore Multimax<sup>TM</sup> and Apple Macintosh<sup>TM</sup> IIx's. Ports to the Intel Hypercube<sup>TM</sup>, AT&T 6386 Work Group System<sup>TM</sup>, and Hewlett-Packard Spectrum<sup>TM</sup> are underway.

### 3 Results

While it is difficult to evaluate the impact of a programming language or a style of programming on program performance, overall, it is the ease with which it is possible to construct an efficient software system that matters. Knowledge of the code that is produced by the compiler for particular programming constructs, the ability of the programmer, and the use of improved algorithms and data structures can mask the advantages or disadvantages of the language or style. Our evaluation of the source code for *Choices* will therefore be presented on its own without attempting to compare it to UNIX. We will analyze the effect of object-oriented design on code structure and reuse, including measurements of the size and depth of the class hierarchy, the lines of code and the number of methods per class, the number of methods inherited by subclasses, and the number of classes used across *Choices* layers and subsystems.

The calibration of our performance measurements is also complicated by the uniqueness of the design of *Choices* as an object-oriented system. But, like any operating system, *Choices* provides certain basic functions. We have therefore selected to measure the performance of attributes of *Choices* that have a close resemblance to features in UNIX and ignored other features. The measurements were made on an Encore Multimax shared



Function, Proxy, and System Call Overheads	
type	overhead in $\mu s$
C function call	3
C++ virtual function call	6
<i>Choices</i> proxy object call	79
<i>Choices</i> system call	81
UNIX system call	98

Table 1: Overhead of Function and System Calls

memory multiprocessor using the 10MHZ NS32332<sup>TM</sup> processor and a microsecond timer. The UNIX system used is Encore's 4.2 BSD UNIX system which includes optimizations for parallel processing. The reader should be aware that a direct comparison between *Choices* and UNIX may not always be meaningful. For example, we use modern algorithms that could be used to replace the older algorithms in UNIX to improve performance. We also use operating system implementations and features that may have been inappropriate at the time UNIX was designed because of the hardware technology then available. However, comparisons to UNIX can be used to put the *Choices* performance numbers in perspective.

Although our results do not allow one to infer the likely performance of an application running under *Choices*, nor that C++ and object-oriented programming is superior to C and standard coding practices, they do show that there is no inherent loss in performance. We believe that this result, when coupled with the other advantages of building object-oriented systems using object-oriented design, is significant.

The rest of this section includes the measurements of: procedure and method call overhead, system and proxy object call overhead, time-slicing overhead, various context switching times, page faulting handling time, memory allocation and deallocation times, and file read, write, open, create, close, and copy times. Before each measurement, we indicate why the measurement is important and how it was made. After each measurement, we discuss how the results should be interpreted.

### 3.1 Procedure Call Overhead

The first measurements we present provide the basis for some later comparisons and analysis. Table 1 compares the overhead between a C function call and a C++ virtual function call. The functions called in this table all take no arguments and return only a single integer. The measured overhead includes both the invocation and return of the call. The C++ function call is slower because of the extra indirection caused by the method lookup in the object's virtual function [14] table and the adjustment of the this pointer to account for multiple inheritance.

Table 1 also compares two different approaches implemented in *Choices* for transferring control from user to system protection mode. These mechanisms allow user programs to request system services, much in the same way as the system call operates under UNIX. The examples shown implement a function that is similar to the "getpid()" system call in UNIX.

The first approach, which uses a proxy object call [13], allows a user program to invoke an operation on a system object by invoking a similar operation on a proxy object that

Time-slice and Time-slicing Overhead	
System function	overhead in $\mu s$
<i>Choices</i> time-slice (system + hardware)	234
<i>Choices</i> minimum time-slice	17
UNIX time-slice (system + hardware)	356

Table 2: Time-slicing Measurements

is in a user read-only region of virtual memory. This is the mechanism by which *Choices* applications request system services. The proxy object method causes a trap into supervisor state and, therefore, *Choices* kernel code. The kernel code validates the call and invokes the appropriate method on the intended system object.

The second approach measured emulates a UNIX system call. It is implemented by trapping into supervisor state and kernel code with a register containing the index of the service requested. This is the mechanism by which a UNIX compatibility library, which is designed to support migration to *Choices* from UNIX, requests *Choices* system services. The last entry in Table 1 shows the overhead for the UNIX `getpid()` system call, placing the previous two numbers in perspective.

The better performance of the proxy object call is a result of exploiting knowledge of the C++ virtual function calling convention. The implementation of the proxy object call avoids saving unnecessary context during the transfer from non-privileged to privileged execution. Both the two *Choices* schemes and the UNIX system call require a small amount of assembly code to be used in their implementation. The UNIX system call implementation does not contain a significantly different amount of assembly code and the `getpid()` service imposes about the same amount of overhead in either UNIX or *Choices*.

### 3.2 Time-slicing Overhead

Both UNIX and *Choices* are timesharing operating systems. Timesharing is implemented by running each program in the ready queue for a quantum of time called a time-slice. Programs are selected from the ready queue using a round robin discipline. Table 2 shows the overhead for implementing a time-slice. The measurements were acquired by time-slicing the execution of a single program that executes a loop and increments a counter 10 million times. For both *Choices* and UNIX, the timings were made on an otherwise idle system (i.e., the ready-queue was otherwise empty). Decreasing the size of the time-slice increases the running time of the program because of additional overhead. The additional overhead is created by the extra time spent trapping into the operating system when the timer expires, deciding which process to execute next, and finally rearming the timer and dispatching the new process. Using an equation relating the elapsed running time to the number of time-slices[13], and the running time with no time-slicing,<sup>1</sup> the measurements can be used to solve a set of simultaneous equations yielding the overhead per time-slice. Under *Choices*, the smallest time-slice that will allow the timer to be set and the user code to be resumed is also shown in Table 2.

The difference between the measurements can be explained by the implementation of

<sup>1</sup>Unlike *Choices*, time-slicing cannot easily be disabled in Encore UNIX.

Process Switching Overheads in $\mu s$				
Process type	Domain	System	Application	FP Application
System	same	88	163	176
Application	same	163	221	—
FP Application	same	176	—	244
System	different	—	289	315
Application	different	289	370	—
FP Application	different	315	—	412

Table 3: Process Context Switching Overhead

interrupts in *Choices*. In *Choices*, all interrupts are handled by immediately resuming a system coroutine which decides what to do next[13]. In the case of an empty ready-queue in *Choices*, the minimum overhead is the cost of the interrupt because the interrupted process will be resumed immediately when the system coroutine determines there are no other processes ready to run. The context switch between application and system coroutine and back from system coroutine to application requires a minimal amount of information to be saved and restored. Further, the interrupt handler invoked for the timer interrupt is used to reset the timer. The small minimum allowable time-slice is possible because the interrupt handler delays starting the timer until it has performed all its other book keeping.

To estimate the approximate cost of time-slicing when the ready-queue is not otherwise empty, the overhead above should be added to the cost of switching between processes discussed in Section 3.3.

### 3.3 Context Switching Overhead

Both UNIX and *Choices* support concurrent processes. Timesharing provides the abstraction that the concurrent processes execute at the same time. Under both Encore UNIX and *Choices* concurrent processes may also execute in parallel on the multiprocessor hardware. Often a task or user application will consist of several communicating concurrent processes. A process may block waiting for synchronization with another process. In this section we examine the overheads associated with concurrent processes.

*Choices* supports lightweight processes. An abstract Process class defines the notion of a process and subclasses define the behaviors of various specializations. Blocking and time-slicing are implemented using context switching primitives. These save and restore the processor state corresponding to a blocked or suspended process. In *Choices*, the context switching primitives used when a process is blocked or suspended depend on the classes of both that process and of the process that will next be run on the processor. When a system process is blocked and another system process is run, minimal context switching occurs. However, when an application using floating point is blocked and a different application is run, a larger overhead is incurred.

The *Choices* process differs from a UNIX process. Under UNIX, a process executes in its own virtual memory. Context switches between processes have a significant overhead and are "heavy-weight". To alleviate this problem, a separate "light-weight" thread package can be used in some systems. The package is not part of standard UNIX. Other

Virtual Memory Overhead	
system	overhead in seconds
<i>Choices</i> 1024, 4k pages	12.1
UNIX 1024, 4k pages	11.6

Table 4: Allocating and Zero Filling Pages

UNIX derivatives have kernels that include support for “heavy-weight” (normal UNIX) and “light-weight” processes. In *Choices*, it is the context switch that is light- or heavy-weight, not the processes themselves.

Table 3 shows some of the context switching overheads of *Choices* for different classes of process. The measurements were made between two processes using a loop in which each process relinquishes the processor to the other process. Unfortunately, it was impossible to repeat this experiment under UNIX since it does not provide primitives to relinquish the processor directly to another process. Also, since the source code of UNIX on the Encore Multimax was unavailable, we could not instrument UNIX to gather a similar performance measure.

Each *Choices* process executes within a virtual memory called a Domain. Several processes may share the same domain. The time for a context switch from one system process to another in the same domain is 88 microseconds. Such a context switch only requires the saving and storing of CPU registers used by the system code. The time for a context switch between two application processes that do not use floating point running in the same domain is 221 microseconds. The full set of registers used by an application is saved. When floating point is used, the overhead increases to 244 microseconds as a result of saving and restoring the floating point registers. The context switch between floating point application processes in different domains is 412 microseconds. The additional overhead incurred when the domain differs corresponds to flushing the MMU cache and reloading the page tables for the new domain.

### 3.4 Virtual Memory Overhead

Both *Choices* and Encore UNIX have a paged virtual memory. In a paged virtual memory, fixed sized “pages” of addresses in virtual memory are bound to blocks or “page frames” of physical memory. The contents of virtual memory are stored on a backing store until they are needed by a program. A page fetch transfers a page of data from backing storage to physical memory. On our Encore Multimax, both UNIX and *Choices* have the same overhead of 25 milliseconds for a page fetch. Details of the *Choices* implementation of virtual memory are discussed elsewhere [12]. The overhead is dominated by the access time of data on the disk used for backing storage.

The disk overhead can be eliminated by considering the allocation and filling of new pages in a virtual memory. Here an application references data that is not yet stored on a backing store. The system creates the data being referenced by allocating physical memory to the page being addressed and filling that page with zeros.

In the measurements shown in Table 4, four megabytes of virtual memory are created and zeroed by accessing the first word of each page. In both the *Choices* and UNIX implementations, the page size is 4096 bytes, thus both systems create 1024 pages. The results show that the *Choices* overhead is nearly the same as UNIX, even though the



Memory Allocation times in $\mu s$		
Operation	<i>Choices</i>	Encore UNIX
Allocate	34	54
Free	39	16
Total	73	70

Table 5: Memory Allocation Measurements

*Choices* code has yet to be tuned.

### 3.5 Memory Management Overhead

While dynamic memory management is important in C[6] programming, it is essential to object-oriented programming in C++. In C, the usual memory management operations are `malloc` and `free`, whereas in C++, they are `new` and `delete`. Using the `new` and `delete` operations, we measured the average overhead for creating and deleting objects of various sizes ranging from 32 to 4096 bytes. Table 5 contains the results of our measurements. Both systems use a similar algorithm for memory management, and both systems incur similar overhead. However, the *Choices* allocator adds several important features. These features include alignment, space efficiency, and support for execution on parallel processors. The UNIX allocator aligns every object on an eight-byte boundary; however, it never aligns objects on page boundaries. The *Choices* allocator aligns all objects larger than one page on page boundaries. Such an alignment policy supports more efficient usage of virtual memory and direct-memory-access (DMA) devices. The UNIX allocator stores state information immediately preceding the blocks it allocates. This extra information, combined with the allocation algorithm, results in twice as much storage being reserved for objects whose sizes are powers of two, sizes that are common in computer systems. On average, the UNIX allocator uses 50 percent more space than the *Choices* allocator. The UNIX allocator was written to be run within a single process. The *Choices* allocator was written to support the requests of multiple processes running on multiple processors; therefore, it uses spin-locks to prevent corruption of its internal data structures by simultaneously executing processes. Almost half of the time the `new` and `delete` operations take in the *Choices* allocator is spent locking and unlocking these data structures. By optimizing the other aspects of memory allocation, the *Choices* allocator achieves similar performance to the UNIX allocator, even though it supports parallelism.

Since C++ does not support automatic garbage collection of unneeded objects, *Choices* uses reference counts. Because the reference and unreference operations are performed extremely frequently within the *Choices*, we optimized their performance. Currently it takes only 25 microseconds to reference and unreference an object. When an object's reference count reaches zero, the object is automatically deleted.

### 3.6 File System Performance

*Choices* provides stream-oriented file systems that conform to operating system standards such as 4.3 BSD UNIX, System V UNIX, and MS-DOS. The file system class hierarchy also supports the construction of customized and experimental file systems. Various instances of file systems can coexist and interoperate in a running *Choices* system. Because

File open, create and close in $\mu s$						
Operation	Cached	Choices		Encore UNIX		
Open existing file	NO	32173	$\pm$ 62	28812	$\pm$	241
Open existing file	YES	4163	$\pm$ 86	2722	$\pm$	14
Open currently open file	YES	2593	$\pm$ 75	2067	$\pm$	86
Create new file	NO	29854	$\pm$ 939	25546	$\pm$	1991
Close file	NO	72208	$\pm$ 2257	80303	$\pm$	22233

Table 6: File System Operation Measurements

the BSD file system is the most efficient of the systems that currently can be built from our hierarchy, and because it uses the same on-disk data structures as Encore's version of UNIX, we have chosen to measure its performance.

Within a disk-based computer system, disk latencies dominate file operation times. To reduce these delays, file systems use various *caching* techniques, such as the *buffer cache* used in UNIX[2] and the *memory-mapped files* used in *Choices*. A buffer cache allows the file system to keep copies of many of the most recently used disk blocks in physical memory. Since recently accessed blocks are more likely to be reused, the cache can greatly reduce the cost of reading and writing data blocks. The UNIX buffer cache is implemented in software. It uses a least-recently-used buffer replacement algorithm and hashing to map disk block numbers to buffer addresses. In contrast, *Choices* allows the file system to reuse the page replacement algorithms of its virtual memory management system. Instead of using a software mapping, *Choices* uses the virtual memory hardware to map requests for disk blocks to buffer addresses.

Because caching often speeds up file operations by a factor of ten, we measured operations both when the operation generated a cache *miss* and a cache *hit*. To make comparisons more significant, we used the same amount of physical memory, two megabytes, for caching disk blocks in both systems, and we tested the operations in the same order on each system. Also, because disk latencies vary from access to access, we repeated each test several times and report the mean value of each measurement and the 95% confidence interval for the mean.

To use the file system, an application program must first gain access to files via *open* or *create* operations. Table 6 contains measurements of the time it takes to open existing files and create new files. The open operation uses both the current directory to convert a file name to an *inumber* and an in-core *inode* to convert the *inumber* to a reference to an open file object. *Choices* takes slightly longer than UNIX to open files, regardless of whether the disk block describing the file is cached. The reason *Choices* takes longer is that it builds a caching object that it uses to map the file into memory. We expect that this small amount of extra overhead for file opens will be amortized over the entire time the file is open. The create operation is similar to the open operation; we chose to only measure an uncached create, since *Choices* flushes modified directory blocks to the disk after a file is created. Again, the creation time of a caching object accounts for the difference between creating files for *Choices* and for UNIX. We also measured the close operation for a file opened in *read-only* mode. The times are similar for both systems. The reason the close operation takes 70 to 80 milliseconds is that the in-core *inode* structure must be written back to the disk, even if the file has not been modified.

Read, write, and lseek times in $\mu s$					
Operation	Cached	<i>Choices</i>		Encore UNIX	
Read block direct	NO	26803	$\pm$ 420	33002	$\pm$ 1275
Read block direct	YES	2524	$\pm$ 106	3784	$\pm$ 128
Read block indirect	NO	58841	$\pm$ 4876	53457	$\pm$ 769
Read block indirect	YES	2726	$\pm$ 294	4358	$\pm$ 219
Write block direct	YES	3752	$\pm$ 207	3884	$\pm$ 324
Write block indirect	YES	3168	$\pm$ 23	4324	$\pm$ 306
Lseek	—	111	$\pm$ 5	194	$\pm$ 6

Table 7: File Operation Measurements

The most important operations on open files are read and write. Measurements of these operations are given in Table 7. Before blocks can be read or written, logical block numbers must be mapped to physical blocks numbers using the data stored in an inode structure. Inodes organize this block mapping information into a variable level tree.<sup>2</sup> We measured I/O operations using both direct blocks and for single-indirect blocks.<sup>3</sup> All the I/O measurements reported in Table 7 are for reads or writes of 8192-byte *aligned* blocks. For cached read and write operations, *Choices* performs better, since it uses virtual memory hardware to map disk block number to buffer addresses. For uncached read and write operations, *Choices* and UNIX perform similarly, since both systems must perform disk I/O and update mapping information.

The lseek operation, which repositions the stream file location pointer, is essential for randomly accessed files. Table 7 also reports the overhead of the lseek operation. *Choices* performs lseeks faster primarily because it provides a more efficient system call mechanism (see section 3.1).

The interactions between various file system operations can often lead to unanticipated results. Therefore, we not only measured the times of individual operations, but we also measured the time of performing a common series of operations: copying an entire file. For this test we chose to copy a one megabyte file; we measured both the time to copy the data blocks from disk-to-disk and from cache-to-cache. Table 8 shows the results of these tests. For disk-to-disk copies, *Choices* performs slightly faster, largely owing to the efficiency of the *Choices* caching mechanism. For cache-to-cache copies, *Choices* takes less than half the time, again owing to the efficiency of the *Choices* caching mechanism. *Choices* also provides a single operation, copy, to copy an entire file. By avoiding the overhead of making many (256) system calls, *Choices* provides a substantially faster file copy mechanism.

## 4 Code Structure and Reuse

*Choices* is written as an object-oriented system. It has 281 classes of which 190 are subclassed from class Object and 90 are miscellaneous support classes. Many of the 90

<sup>2</sup>In BSD UNIX, this tree can have 0, 1, or 2 levels of indirection. In System V UNIX, this tree can have up to 3 levels of indirection.

<sup>3</sup>Double and triple-indirect blocks are seldom used.

Copy one megabyte files in seconds			
Block size	Cached	Choices	Encore UNIX
8192	NO	8.167 $\pm$ 0.02	8.542 $\pm$ 0.11
8192	YES	.906 $\pm$ 0.02	2.019 $\pm$ 0.11
1048576	YES	.562 $\pm$ 0.03	—

Table 8: File Copy Measurements

Classes and Methods in Levels of <i>Choices</i> Hierarchy			
Level	Classes	Public Methods	Protected Methods
1	1	14	0
2	37	197	104
3	56	386	71
4	46	304	60
5	31	143	98
6	18	122	63
7	2	8	1
All levels	191	1174	397

Table 9: Class Hierarchy Characteristics of *Choices*

classes are introduced to hide machine dependent details. Most of the 190 subclasses are specializations of several abstract classes. Table 9 shows the number of classes that are subclassed from class Object and the depth that they occur within the *Choices* hierarchy. The average depth of a class within the hierarchy is 3.7. Only 10% of the classes have a depth greater than 5. The results support other evidence that inheritance has been used within *Choices* to structure and reuse the code. The performance measurements of *Choices*, as shown in the previous section, indicate that any overheads incurred by method lookup and the object-oriented organization of the system appear to be trivial compared with the use that was actually made of subclassing in the construction of the system.

Table 10 shows the organization of *Choices* code into files of C++ and assembler. Just over half of the files used by *Choices* contain header information. Less than 11% of the files contain machine or processor dependent information. Table 11 shows the organization of *Choices* in terms of the lines of code written in C++ and assembler. Assembler code represents 0.7% of the total number of lines of code, and this is isolated to just ten files. Header lines of code represent 34% of the total number of lines of code. The large number of header files in comparison with the number of lines of code they contain is

Files of C++ and Assembler Code in <i>Choices</i>			
Type	“.h” files	“.c” files	Total files
Entire <i>Choices</i> source code	286	232	518
Containing machine dependent code	13	19	32
Containing processor dependent code	10	12	22
Containing assembler code	0	10	10

Table 10: Source File Characteristics of *Choices*



Lines of C++ and Assembler Code in <i>Choices</i>			
Type	“.h” lines	“.c” lines	Total lines
Entire <i>Choices</i> source code	27,012	51,768	78,780
Machine dependent code	2,200	5,635	7,835
Processor dependent code	1354	4,434	5,788
Assembler code	0	555	555

Table 11: Code Characteristics of *Choices*

a consequence of trying to organize *Choices* for easier maintenance and less compilation interdependencies.

## 5 Conclusions

Despite the scarcity of support tools, C++ is a powerful language in which to prototype high-performance systems using an object-oriented approach. The characteristics of object-oriented systems have reduced the effort needed to make major changes in the design and class hierarchies of these prototypes. Classes and inheritance have yielded much code reuse, both within and between components and subsystems. Several developers have been able to work together while avoiding some of the usual pitfalls inherent in large software projects.

Our evidence shows that the benefits of using an object-oriented language for coding system algorithms outweigh the slight overhead of C++ method calls. Many of the *Choices* subsystems outperform and none are significantly slower than their UNIX equivalent, even though the UNIX system used for comparison is optimized for a multiprocessor. Despite the lack of performance analysis tools and the difficulties of making fair comparisons, we encourage others to experiment with building high-performance systems using object-oriented design and programming.

## References

- [1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of Summer USENIX*, July 1986.
- [2] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.
- [3] Roy H. Campbell, Gary M. Johnston, Peter W. Madany, and Vincent F. Russo. Principles of Object-Oriented Operating System Design. Technical Report UIUCDCS-R-89-1510, University of Illinois at Urbana-Champaign, April 1989.
- [4] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop*, pages 109–123, Santa Fe, New Mexico, November 1987.
- [5] Gary Johnston and Roy H. Campbell. An Object-Oriented Implementation of Distributed Virtual Memory. In *Proceedings of the USENIX Workshop on Distributed and Multiprocessor Systems*, pages 39–58, September 1989.

- [6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1978.
- [7] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming*, pages 311-328, Nottingham, UK, July 1989. Cambridge University Press.
- [8] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference*, Denver, Colorado, October 1988.
- [9] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181-197, August 1984.
- [10] Peter Norton. *The Peter Norton Programmer's Guide to the IBM PC*. Microsoft Press, 1985.
- [11] Vince Russo, Gary Johnston, and Roy H. Campbell. Process Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '88*, San Diego, CA, September 1988.
- [12] Vincent Russo and Roy H. Campbell. Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems using Class Hierarchical Design. In *Proceedings of OOPSLA '89*, New Orleans, Louisiana, September 1989.
- [13] Vincent F. Russo. *The Design and Implementation of an Object-Oriented Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, April 1990 (in preparation).
- [14] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986.
- [15] K. Thompson. Unix implementation. *Bell System Technical Journal*, 57(6):1931-1946, July 1978.

# RIPE: An Object-Oriented Robot Independent Programming Environment\*

David J. Miller  
and  
R. Charleene Lennox

Sandia National Laboratories  
P. O. Box 5800  
Albuquerque, NM 87185

## ABSTRACT

An object-oriented Robot Independent Programming Environment (RIPE) developed at Sandia National Laboratories is being used for rapid design and implementation of a variety of applications. A system architecture based on hierarchies of distributed multiprocessors provides the computing platform for a layered programming structure that models the application as a set of software objects. These objects are designed to support model-based automated planning and programming, real-time sensor-based activity, error handling, and robust communication. The object-oriented paradigm provides mechanisms such as inheritance and polymorphism which allow the implementation of the system to satisfy the goals of software reusability, extensibility, reliability, and portability. By designing a hierarchy of generic parent classes and device-specific subclasses which inherit the same interface, a Robot Independent Programming Language (RIPL) is realized. Work cell tasks demonstrating robotic cask handling operations for nuclear waste facilities are successfully implemented using this object-oriented software environment.

## 1. INTRODUCTION

This paper discusses the Robot Independent Programming Environment (RIPE) developed at Sandia National Laboratories. RIPE is an object-oriented approach to robot system software architectures. The primary accomplishment of this effort is a software environment which facilitates the rapid design and implementation of complex robot systems to support diverse research efforts and applications. RIPE allows robot system developers to concentrate on algorithm design and optimization, as well as testing and evaluation of new control, sensing, computing, and communications technologies without having to focus on overall system software development and integration. This is achieved by modeling the robot system as a set of software classes. As a result, RIPE hides device integration details and provides uniform interfaces to all objects in the system. A separation of concept from implementation characterizes RIPE's software classes and provides software reusability, extensibility, reliability, and portability.

In the following sections, the problems associated with complex robot software systems (which we have experienced first hand) are reviewed, together with past approaches dealing with these problems. We then discuss the underlying object-oriented concepts and distributed computing architecture upon which RIPE is based. RIPE currently supports automatic motion planning and programming of robotic and machining devices based on models of the environment, sensor-based control, error handling, and robust communication. The RIPE architecture also supports development of advanced software concepts such as graphical interfaces for robot system control.

The detailed design of RIPE is then defined, followed by a discussion of two implementations involving robotic cask handling operations for nuclear waste facilities. These systems show how the class hierarchy, consisting of generic parent classes and device-specific subclasses sharing the same interface, results in a Robot Independent Programming Language (RIPL). Finally, a brief discussion of future work is presented.

## 2. CURRENT APPROACHES TO ROBOT SOFTWARE

Sandia is currently developing robot systems for applications including hazardous material handling, automated assembly, and robotic edge finishing. Supporting this work are research laboratories investigating controls and optimization, telerobotics, grasping and dexterous manipulation, vision, tactile and proximity sensing, path planning and collision avoidance, oscillation damped movement, autonomous vehicles, flexible arms, and simulation. The diversity of Sandia's robotics effort implies that the software environment must support a wide variety of requirements and devices. It also must serve users with different levels of expertise in robot system programming.

### 2.1 The Problems

As our robotics effort has developed we have experienced many of the problems common to robot programming:

- the inability of robot languages to handle integration of sensors into the motion control system
- the difficulty of extending application code to include new tasks or new devices
- the high expense of application programming
- the small amount of code that is reusable for new systems
- the focus on manipulator-level rather than on task-level programming
- the management of system complexity

\*This work was supported by the U. S. Department of Energy under Contract DE-AC04-76DP00789

- the time consuming process of writing and debugging low-level (e.g. communication) software before beginning to test new algorithms
- the need for real-time control of mechanical devices
- the necessity of implementing robust error handling and recovery code

It has been our experience that software development costs are a significant part of the overall intelligent robot system development costs. Finding solutions to robot software problems reduces these costs and thereby increases the viability of using intelligent robot systems in applications where it was once considered too costly.

## 2.2 Other Approaches to Solving the Problems

Various approaches have been used to try to solve these problems. Vendor supplied robot programming languages have been enhanced until they resemble general-purpose, high-level computer languages[15][11]. However, applications written in these languages are specific to the vendor's robot and, therefore, are not portable.

One approach is to replace the robot control system with a new control system embedded within an existing operating system, written in a general purpose language, and using primitive functions included in a library[6][1]. Frequently the resulting system is not a viable option for industry work cells or research labs since it requires the replacement of the robot control system hardware.

A number of the more recent approaches [7][16][17] use conventional programming languages to help solve the problems of system complexity, real-time constraints, sensor integration, and modeling. These approaches have the advantages gained by using a language which is reliable, well-supported, portable, and familiar. In addition, these approaches focus on the total work cell system rather than only on the robot. The drawback is that, to date, the software has often been single purpose and not easily extended to other applications.

## 2.3 Approach for RIPE

RIPE was developed to support model-based automated planning and programming of robotic and machining devices, integration of sensor technologies, development of next generation robot system programming languages with graphical interfaces, error handling, and robust communication. It is built upon well-established software operating systems and programming languages. RIPE is an environment for complex system integration which stresses use of off-the-shelf hardware (e.g. commercial robots) where appropriate as well as providing support for advanced system development.

### 2.3.1 Computing Architecture

The RIPE computing architecture consists of a hierarchical multiprocessor approach which employs distributed general and special purpose processors. This architecture provides the computing power required by the RIPE software to control complex diverse subsystems in real-time while coordinating reliable communications between them. Advances in microprocessor technology allow general purpose processors to achieve the computing performance required by complex robot control algorithms while remaining compatible with a large base of existing software.

### 2.3.2 Object-Oriented Design

Robot systems perform actions on certain objects within a defined work space. The software controlling the robot system can be viewed as an operational model of the world in which the robot exists. Therefore, RIPE is organized around representations of the objects in the work space so that its structure reflects the physical structure of the system. Controlled complexity is achieved by creating, combining, and manipulating software objects instantiated from previously defined software classes to perform the specific tasks of the system.

Abstract data types, classes, and objects allow the designer to model the physical robotic work cell entities in RIPE by defining only their attributes, behavior, and interfaces. Examples of such entities include work pieces or parts with geometric attributes, devices (e.g. NC machines, fixtures, machine tools, tool changers, robots, grippers, other end effectors), and sensors for contact switches, force control, and vision. Since the software classes in RIPE are defined to represent the physical objects that are commonly found in a work cell, calls to member functions of these generic software classes become the general device independent language used to program the cell. This is achieved by applying polymorphism which enables objects of a generic parent class and objects of its device-specific subclasses to receive the same messages and respond to them appropriately. The device independent programming language in RIPE resulting from object-oriented design of robotic work cells is RIPL.

Separating robot system concepts from the actual RIPE implementation results in robot system software which is reusable, extensible, reliable, and portable. RIPE's reusability is the basis for the design process. Extensibility is provided in RIPE by defining new software classes which in turn become part of the general work cell programming language. Use of inheritance to define subclasses which are extensions or restrictions of RIPE parent classes greatly lowers the cost and complexity of software development. System reliability is enhanced by reusing well-defined RIPE objects, and portability is realized because RIPE classes are tightly encapsulated and relatively independent of their environment.

### 2.3.3 The Development Environment

Our development environment for RIPE has four primary layers: task-level programming, supervisory control, real-time control and device drivers. The choice of software at each layer is influenced by the primary requirements for modeling, sensing, and motion specification, as well as the widely acknowledged levels of robot software (task, manipulator, servo) [2, 18]. In addition, there is a strong relationship between the architecture employed at each particular layer and robot performance requirements.



### CASK HEAD WORK CELL CLASS HIERARCHY

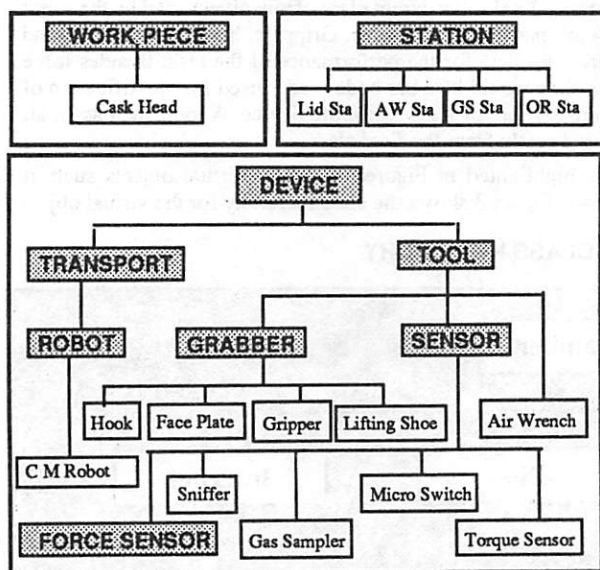


Figure 1

### RADIATION SURVEY WORK CELL CLASS HIERARCHY

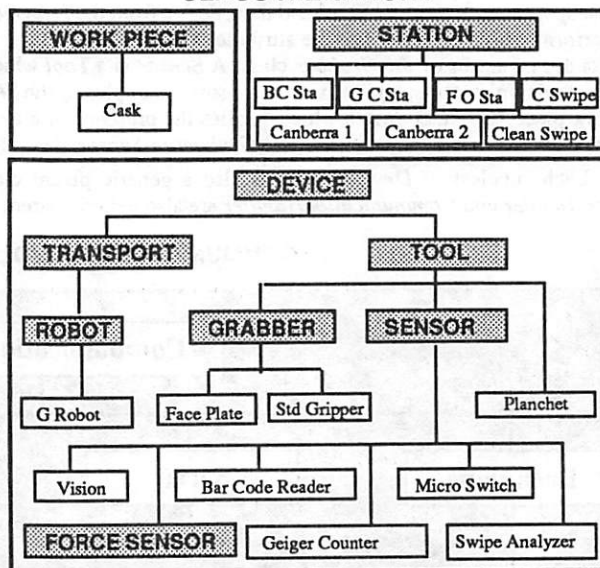


Figure 2

The first layer is synonymous to what is generally referred to as task-level programming. At this level, world modeling, planning, and simulation are performed. Currently, this layer is in the initial stages of definition in our architecture.

The second layer is the supervisory control layer implemented on a UNIX-based workstation. This layer contains the primary control programs which coordinate all devices and activities of the system. The C++ language [13] is used to implement the object-oriented work cell class hierarchies and the supervisory code which manipulates these classes. Because C++ is a superset of C, a large existing base of C code is used, and all of the advantages of C programming are retained (portability, versatility, and systems programming facilities).

The third layer in the programming environment handles real-time control of devices for tasks such as force control. This layer consists of multiple VME-based 68000 family processors on a backplane network running the VxWorks operating system [19]. VxWorks was selected because of its real-time kernel, full-featured development and run-time environments, and its compatibility with UNIX. C++ runs effectively in this environment, and therefore, the same software can be used both at the workstation level and the real-time control layer. An Ethernet-based local area network ties together the workstations and VME systems.

The bottom layer contains the device drivers for each subsystem in a work cell. Some device drivers are relatively simple and consist of interfacing class members to firmware for tasks such as controlling a bar code reader. Others are sophisticated programming environments, such as the CIMCORP XR100 gantry robot software system.

In the case of intelligent devices such as robot controllers, a monitor program located at the controller for the device is written in the robot programming language. This monitor establishes communication with an external host CPU, waits for a command from that host, carries out the command when one is received, and then waits for the next command. The monitor is treated as the part of a distributed robot object which resides on the robot controller. The messages that it understands and interprets are defined in the robot class and frequently have a one-to-one correspondence with the public member functions defined for the robot class.

## 3. DEFINITION OF ROBOT SYSTEMS IN RIPE

The design of RIPE is based not only on our goals of ease of use, expressiveness, extensibility, and reusability but also on compatibility with FAC-SIM [5], a simulation system developed at Sandia for the analysis of robot systems. The partitioning of a system into classes is fairly straightforward since most classes reflect the physical objects of the application. The software classes which do not represent physical objects are termed "virtual objects" and include *CommunicationHandler*, *WorldModeler*, *ErrorHandler*, *TrajectoryGenerator* and *PathPlanner*.

### 3.1 The Generic Objects

The class inheritance hierarchies in RIPE are designed to allow the programming of tasks using generic classes. Figures 1 and 2 illustrate two example system hierarchies. In the Cask Head Work Cell, a Cincinnati Milacron robot performs leak detection and gas sampling operations on the head of a cask containing nuclear waste. In the Radiation Survey Work Cell, contamination surveys of the cask are performed by a CIMCORP gantry robot. The division of a robot system into the three basic classes of *WorkPiece*, *Station* and *Device* is derived from the concept that devices carry out actions on work pieces, and stations are locations in the work space for storing these devices or work pieces. The definition of the class hierarchy for *WorkPiece* and *Station* is specific to an application, but each work cell has several kinds of devices, e.g. robots, sensors, grippers, and other tools. All devices which carry

out actions are derived from the parent class *Device*. Instead of, or in addition to a manipulator, a system might employ other devices such as an NC machine, a conveyor, a remotely controlled fork lift, or a mobile robot. These have the property of being able to move or transport a work piece or a tool and thus, derive from the *Transport* class. *Tool* is the parent class of any object used by the robot to perform a task. *Grabber* has the attribute of being used to pick up work pieces or other tools. Grippers, hands, face plates, and hooks are instances of the *Grabber* class. A *Sensor* is a *Tool* which provides data for the performance of the task. Besides force sensors, vision systems, and proximity sensors, examples of the *Sensor* class would be a bar code reader used for identification of a work piece, a contact switch which verifies the presence of a tool in its station, or a gas sampling device. A tool, such as an air wrench, which is not an instance of the *Grabber* or *Sensor* class, derives directly from the *Tool* class.

Each subclass of *Device* which is also a generic parent class is highlighted in Figures 1 and 2. Virtual objects such as *ErrorHandler* and *CommunicationHandler* are also generic parent classes. Figure 3 shows the class hierarchy for the virtual object

### COMMUNICATION HANDLER CLASS HIERARCHY

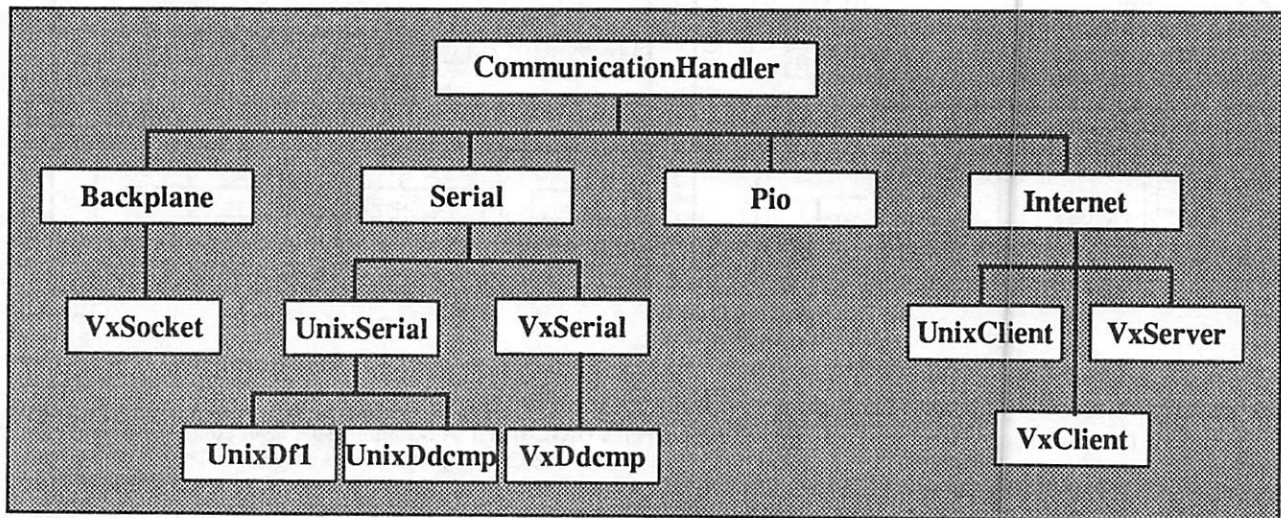


Figure 3

*CommunicationHandler*. The routines that define the user interface to a generic class, along with other attributes and routines common to all instances of that class, are defined at the generic parent level of the hierarchy.

Figure 4 shows the definition of the *Robot* class. The attributes are defined in the data structures such as *current* and *home*. The actions that are common to all robots are found in the definitions of the routines. The keyword *virtual* at the beginning of each function declaration indicates that the routine is to be defined by a device-specific subclass. All robots can be commanded to *move*, but the definition for executing a move is specific to a particular robot. Note the presence of default values for some parameters in the virtual routines, for example *speed* in the *move*, *move\_rel*, and *path move* declarations in Figure 4. Including optional parameters and default values provides flexibility in the subclass definition and in the application. For example, if the user wants to specify a speed during a move, he invokes the desired *move* routine with the *speed* parameter set. If he does not set the *speed* parameter, the default speed, which is established earlier by a call to *set\_speed*, is used.

### 3.2 Derived Objects for Specific Applications

Figures 1 and 2 show how objects in specific work cells are derived from their generic parent classes. The bottom level of the hierarchy enumerates the software representation of the physical objects in the work cell. The Cask Head Work Cell, as represented in Figure 1 for example, required programming of the *CMRobot* class as well as other types of subclasses representing the different devices and work pieces. The interface to *CMRobot* is already defined in parent class *Robot*. The interface serves as a kind of template so that the programming is a matter of "filling in the blanks." For example, the code for the *move* routine consists of translating the command into the format the Cincinnati Milacron controller expects and invoking the *CommunicationHandler*'s routine *send\_msg*.

If a subclass has more capabilities than the parent, the user interface to the subclass is the set of routines defined for the parent class plus additional ones defined in the subclass. The *CMRobot* class has been extended for research into oscillation-damped movement of a flexible beam. Routine *flex\_beam\_damping* uses the *move\_comply* command and torque feedback to actively damp vibrations of a cantilevered beam. When only the routines specified in the generic parent class are used in programming an application, a different derived object representing a different physical device can be substituted, and the commands in the application code remain the same. These routines define the primitives for RIPL.

Figure 2 shows the derived objects for the Radiation Survey Work Cell. A comparison of the two work cells shows that the same set of generic classes can provide very different software object systems. The tasks performed in the application determine which class routines will be invoked in the supervisory code. Thus, changing the way the supervisor uses these class routines can result in the implementation of an entirely different task even though the class definitions remain unchanged.

## ROBOT CLASS DEFINITION

```

class Robot: public Transport
{ protected:
    point    home, current;
    int      current_coordinate_type,
    double   speed;
    int      current_speed_attribute,
    double   accel;
    int      current_accel_attribute;
public:
    // Abstract class so constructor is empty
    Robot();
    virtual ~Robot();
    virtual int move(point loc, int motion_attributes, double speed=0.0);    // Absolute move to loc.
    virtual int move_rel(point delta, int motion_attributes, double speed=0.0);    // Move to a position delta away from current position.
    virtual int move_home();    // Move to home (ready) position.
    virtual void approach(point loc, int axis, double dist);    // Move to a position dist away from loc along axis.
    virtual int depart(int axis, int direction, double dist, double increment=0.0);    // Move along the specified axis distance dist.
    virtual int move_react(point loc, Sensor * FS_ptr);    // Move to position loc under force control.
    virtual int move_comply(point loc, int numinors, Sensor* FS_ptr, double fmax = F_MAX);    // Move to position loc with compliance f.
    virtual int path_move(path_point * p, int motion_attributes);    // Move to each point in path p.
    virtual int path_move_rel(path_point * p, int motion_attributes);    // Move relative to each point in path p.
    virtual void stop();    // Cause the robot to stop.
    virtual int set_speed(double s, int speed_attributes);    // Set the robot member speed to s.
    virtual double get_speed(int speed_attributes);    // Return the value of robot member speed.
    virtual int open_gripper();    // Open the gripper
    virtual int close_gripper();    // Close the gripper
    virtual int get_effector(Device * t_ptr);    // Move to station and pick up effector.
    virtual int put_effector(Device * t_ptr);    // Stow end effector at station.
    virtual int perform(int task);    // Execute a taught sequence
    virtual void where(point cur_loc, int coordinate_type);    // Set cur_loc to the Cartesian position of the robot.
    virtual int report_status();    // Print the current status of the robot.
};

```

Figure 4

## 4. APPLICATION OF RIPE

The Cask Head Work Cell and Radiation Survey Work Cell were constructed as part of the Advanced Handling Technologies Project (AHTP) at Sandia National Laboratories [4]. The AHTP includes efforts to automate cask handling operations at nuclear waste facilities. These work cell prototypes serve as proof-of-concept systems to demonstrate cask handling operations that might be performed robotically.

### 4.1 Cask Head Work Cell Example

The AHTP consists of several subprojects, one of which is the Cask Head Operations (CHO) project. The CHO project investigates robotic performance of cask head operations required before and after nuclear fuel bundle unloading. The Cask Head Work Cell was designed as a prototype system for cask head operations which include leak detection, gas sampling (port cover removal/replacement and coupling/decoupling of the sampling apparatus at the port), and bolting and unbolting operations. Robust algorithms were required for mating the torque wrench to various bolt heads on a cask head mock-up using force feedback. The requirements for the work cell operations illustrate the application of the RIPE environment.

One of the premises on which object-oriented design is based states that designers should avoid as long as possible describing and implementing the specific tasks of a system[9]. Rather, they should produce a high level design that defines only a set of classes which characterizes the behavior of the objects in the system. We followed this principle by designing and implementing the necessary Cask Head Work Cell classes, as discussed above, independent of any application to which they would be applied. As anticipated, implementation of the actual cask head operations was fast and straightforward. All that was required was to create and manipulate the work cell objects to perform the specific tasks of the system. Also, other applications of the work cell such as



flexible beam oscillation damping research [12] were easily implemented because the classes had been designed completely independent of any particular work cell activity.

#### 4.1.1 Computing Environment

Figure 5 illustrates the computing architecture that is used to control the Cincinnati Milacron work cell doing cask head operations.

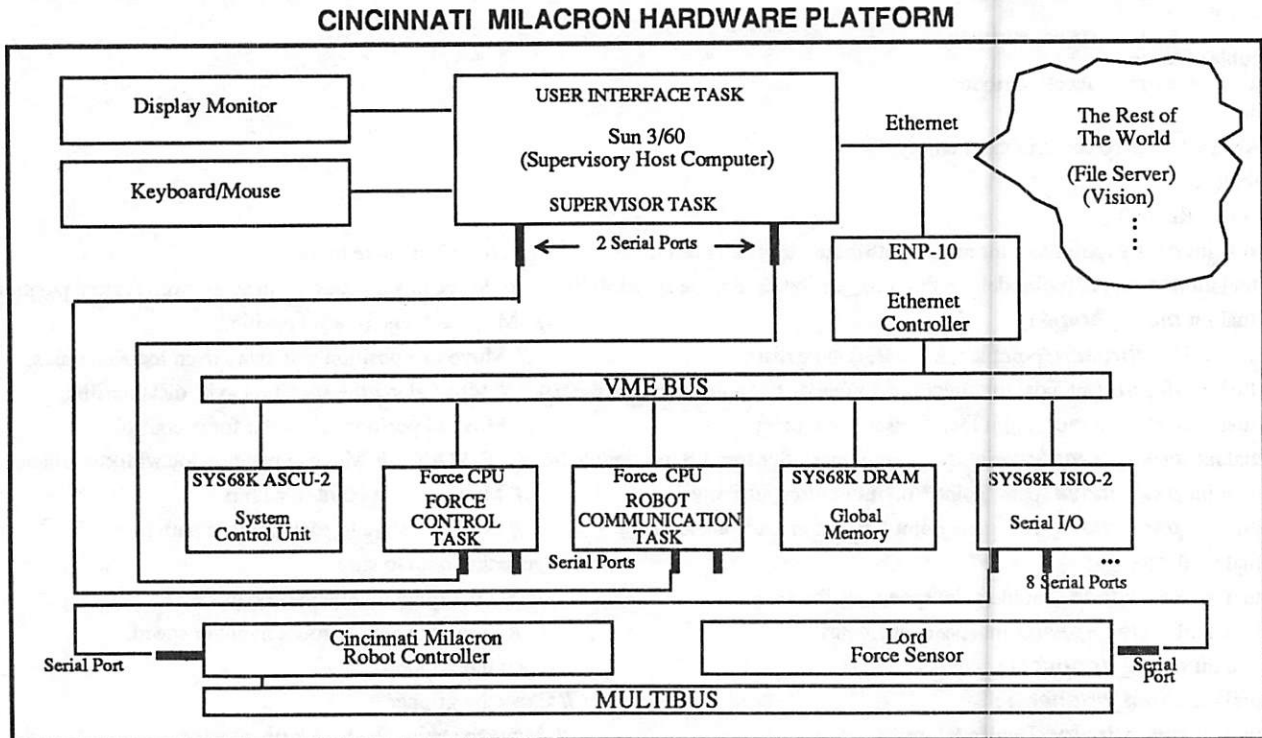


Figure 5

The primary components include a Sun 3/60 workstation, a VME bus with two Force 68020 CPU's, global memory, an 8-port serial I/O card, a Cincinnati Milacron series T3-786 robot, and a Lord force sensor. The computing elements, robot, and force sensor are all commercial subsystems. Special end effectors and grippers were designed and built at Sandia.

The distributed VME multiprocessors coordinated by the Sun workstation allow individual CPUs to control each subsystem in the work cell and provide support for continuous tasks, concurrency, synchronization, data sharing, communications, real-time control, and sensor-based activity. In addition, this architecture reflects the hierarchical layered approach to hardware which corresponds with the different levels of robot software (task, manipulator, servo). However, the design of RIPE allows the software to be mapped onto multiple layers of the hardware, depending upon the application. For example, the servo level normally resides at the robot controller, but whenever compliant motion is performed, some of the servo software functions are executed on the VME bus CPUs. Similarly, the manipulator level software may reside on either the robot controller, VME bus, or the Sun workstation. The task-level software will normally be at the workstation level. Finally, model-based control requires that knowledge about the work cell and its contents be distributed among the software objects that logically represent their physical counterparts, and these objects may reside at any level of the hardware.

#### 4.1.2 Software

To perform cask head operations, four software tasks were required. All of the tasks are implemented in C++ and utilize the communication and device class libraries discussed above to perform their functions. A UNIX environment exists on the Sun workstation, and a VxWorks environment controls the VME hardware. Figure 5 shows how the tasks are distributed among the work cell CPUs for the Cask Head Operation task.

The first task allows the operator to interact with work cell devices. The current implementation uses SunView [14], but future interfaces will be built with an object-oriented package called InterViews [8]. The second task, which also resides on the Sun, serves as the work cell supervisor. It accepts commands from the operator through the first task and carries out these commands by initiating appropriate work cell actions (which may be performed by other hardware components).

The remaining two tasks reside on the VME bus CPUs. One task monitors the Lord force sensor (mounted on the wrist of the Cincinnati Milacron robot) and computes position updates to control robot movement whenever the torque wrench has to mate with a bolt. The other task provides the communications to the robot controller, utilizing the DDCMP protocol [3]. Both tasks use a serial I/O card for message transmission to the force sensor and robot controller.

Figure 6 illustrates the objects that are created by these tasks whenever they are executed. The work cell supervisor creates a *LORDForceSensor* object and a *CMRobot* object. These two objects are, in a sense, distributed over multiple environments. The



## CASK HEAD WORK CELL EXAMPLE

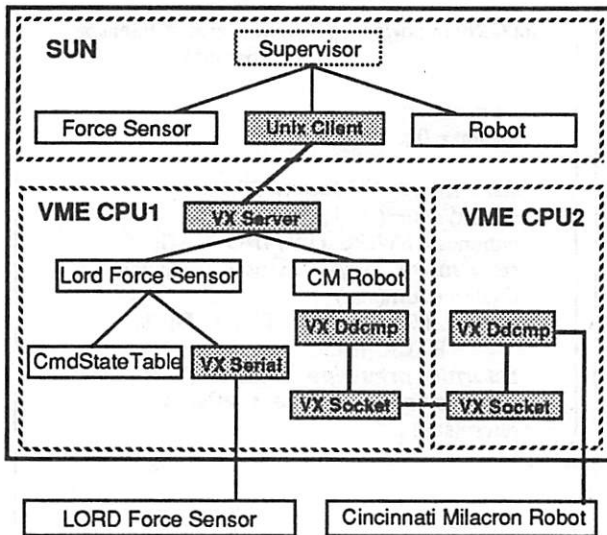


Figure 6

## RADIATION SURVEY WORK CELL EXAMPLE

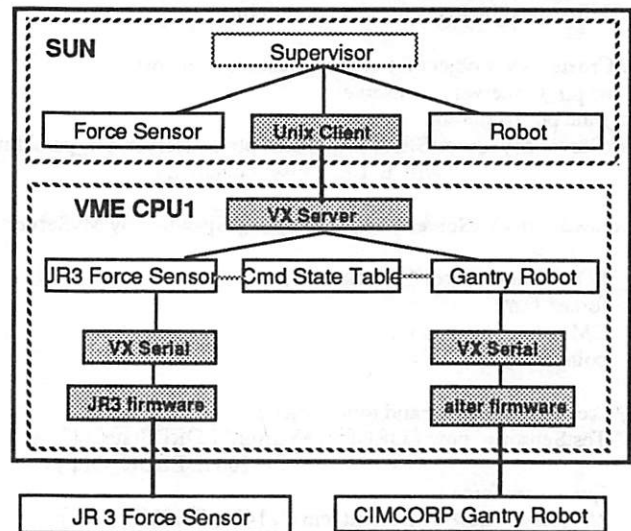


Figure 7

way they are created (argument list specification) determines how they are distributed and how they communicate with the actual devices. In figure 6 the shaded boxes indicate the communication objects created by the device objects.

For example, if the force sensor were to be controlled directly from the Sun workstation, a *LORDForceSensor* object could be created with a parameter list that would cause the creation of a *UnixSerial* object for direct communication to the force sensor device through a Sun serial port. In our implementation for cask head operations, the *LORDForceSensor* object and *CMRobot* object are distributed across both the Sun workstation and a VME CPU due to the real-time requirements of force servo control. They communicate through *UnixClient/VXServer* objects over the Ethernet between the UNIX environment on the workstation and the VxWorks environment on the VME bus. The *LORDForceSensor* and *CMRobot* objects on the VME CPU, in turn, create *VxSerial* and *VxDdcmp* communications objects respectively which allow them to talk to the actual hardware in the work cell. To achieve the update rate necessary for force control, an additional *VxDdcmp* object is created on a second VME CPU to handle the low-level protocol and message transmission to the robot controller. The two distributed *VxDdcmp* objects communicate with each other over the VME bus through a *VxSocket* object.

Finally, the *LORDForceSensor* object also creates a *CmdStateTable* object which reads a configuration file that defines the specific behavior of the Lord force sensor device. For example, using the *CmdStateTable* information, the *LORDForceSensor* object knows that it must send an "OA<CR>" (Output ASCII) command to the force sensor in order to obtain ASCII readings of the current forces being sensed. By isolating device-specific attributes and commands into files that are managed by the *CmdStateTable* object, methods that control the device's behavior are written generically and can reside in the parent *Tool*, *Sensor*, or *ForceSensor* classes rather than in the *LORDForceSensor* class. These methods therefore can be used by other types of force sensor classes derived from the parents (such as the *JR3ForceSensor* class) which have their own configuration files.

Figure 8 illustrates one portion of code for the Force Control Task residing on the first VME CPU. It shows how the objects in the work cell are created and used to perform a simple bolting operation. Messages are sent to the *LORDForceSensor* and *CMRobot* objects to obtain force readings and initiate robot motion until the torque wrench is properly seated on a bolt.

## 4.2 Radiation Survey Work Cell Example

The Radiation Survey Work Cell was the first experimental system to be built for AHTP. Its initial application, the Robotic Radiation Survey and Analysis System (RRSAS), was completed in August, 1987 [10], and included operations to locate a half scale cask mock-up in the work cell using stereo vision, identify cask contents by reading bar codes, and perform both non-contact and contact radiation surveys. Key technologies such as automatic motion planning and programming of the CIMCORP XR100 gantry robot and force sensor integration to maintain constant force contact with the cask surface during contamination surveys are demonstrated by RRSAS. RRSAS was completed prior to the development of RIPE.

Some subsequent studies currently under development include the Impact Limiter Handling project and the Cask Tiedown project [4]. These projects will investigate robotic removal, handling, and replacement of cask impact limiters and tiedowns. They also require technologies similar to those developed in RRSAS, including machine vision and force control. These new projects are being implemented in RIPE.

Although RRSAS was originally implemented in C from a function-oriented top-down design, its highly modular structure and generic functions for robot control made it possible to use some of the existing code for the member functions of the C++ gantry robot class (*GRobot*). Also, because the generic *Robot* class had already been defined and much had been learned during development of the *CMRobot* class, implementation of the *GRobot* class was fairly automatic. Similarly, the Radiation Survey Work

## FORCE CONTROL CODE EXAMPLE

```
//Defines and includes for class definitions
:
// Create server object for communication with host
func ptr_testserver = testserver ;
p_func ptr_panicfunc = panicfunc ;
VxServer MyServer(SERVER_NUM, ptr_testserver, ptr_panicfunc,
    TABLE_LEN, msg_table[0]);
:
int testserver(VxServer* TestServer) // Spawned by MyServer
{
    LORDForceSensor* TestSensor ;
    force* fptr ;
    CMRobot* ptr_robot ;
    point new_loc;

    // Create force sensor and robot objects
    TestSensor = new LORDForceSensor("LORDStateTbl", 4,
        19200, DEBUG_OFF) ;

    fptr = new force ;
    ptr_robot = new CMRobot("cm1", 1401, DEBUG_OFF) ;
    :
    // Wait for a command from the client
    len = TestServer->receive_msg(line, MAX_SOCKET_MSG+1,
        LU_MONITOR);
    :
    // Initiate robot communication & configure sensor
    ptr_robot->perform(WAIT_FOR_BEGIN_REMOTE);
    TestSensor->set_bias() ;
    TestSensor->set_output_mode(BINARY_MODE) ;
    table_entry = ptr_robot->report_var_entry(HOST_ENTRY);

    // Select a work cell activity
    switch(table_entry)

    // Feel for bolt with torque wrench until they mate
    case 1:
        while (i < 22) {
            new_loc[Z] = -0.150; new_loc[R] = 0.0 ;
            printf("MOVE DOWN -0.150 INCHES\n");
            ptr_robot->move_rel(new_loc);
            TestSensor->take_reading(fptr) ;
            if (fptr->t_gain[Z] > -10.0) {
                printf("SUCCESS IS TRUE\n");
                success = TRUE ; break ;
            }
            else {
                new_loc[Z] = 0.150;
                printf("MOVE UP 0.150 INCHES\n");
                ptr_robot->move_rel(new_loc);
                new_loc[Z] = 0.0 ; new_loc[R] = -5.5 ;
                printf("ROTATE 5.5 DEGREES\n");
                ptr_robot->move_rel(new_loc);
                i++;
            }
        }
    :
} // End this work cell activity
```

Figure 8

## SWIPE OPERATION CODE EXAMPLE

```
int GRobot::swipe_operation (Sensor* FSensor,
    float inc)
{
    int numincs;
    int stat = 0 ;

    stat = move_till_touch (FSensor);
    if (stat) return(stat) ;
    numincs = SWIPE_DIS / INC_MAG;
    stat = make_swipe (FSensor, inc, numincs);
    if (stat) return(stat) ;
    depart (FORCE_AXIS, BACK_DIST,
        BACK_INC);
    returnto_preswipe_location();
    clear_offsets_and_zero_alter();
    return(stat) ;
}

int GRobot::move_till_touch (Sensor* FSensor)
{
    point loc ;
    int stat = 0 ;

    FSensor->desired_values[Y] = TOUCH_SET ;
    loc[X] = 0.0 ;
    loc[Y] = APPRO_INC;
    loc[Z] = 0.0 ;
    loc[D] = 0.0 ;
    loc[E] = 0.0 ;
    loc[R] = 0.0 ;
    stat = move_react (loc, FSensor) ;
    return(stat) ;
}

int GRobot::make_swipe (Sensor* FSensor,
    float inc, int numincs)
{
    point loc ;
    int stat = 0 ;

    FSensor->desired_values[X] = 0.0 ;
    FSensor->desired_values[Y] = SWIPE_FORCE ;
    FSensor->desired_values[Z] = 0.0 ;
    FSensor->desired_values[D] = 0.0 ;
    FSensor->desired_values[E] = 0.0 ;
    FSensor->desired_values[R] = 0.0 ;
    loc[SW_AXIS] = inc ;
    stat = move_comply (loc, numincs, FSensor) ;
    return(stat) ;
}
```

Figure 9

Cell employs a JR3 force sensor rather than a Lord force sensor, so a *JR3ForceSensor* class also had to be implemented. This again was facilitated by the existence of generic parent classes (*Tool*, *Sensor*, *ForceSensor*).

#### 4.2.1 Computing Environment

Aside from the specialized subsystems required by RRSAS, the primary hardware components of the Radiation Survey Work Cell are the same as those found in the Cask Head Work Cell (Sun workstation, VME bus, robot, sensors). There is of course a different robot and a different force sensor, and the VME bus uses Heurikon 68020 CPUs rather than Force CPUs (transparent to the software). As stated earlier, this hierarchical distributed approach is our standard architecture which provides the power, compatibility, flexibility, and extensibility needed to implement complex work cell environments.

#### 4.2.2 Software

The first application using RIPE in the Radiation Survey Work Cell performs force controlled movement of the robot arm for the random contact swipe survey. The original force servo control system in RRSAS consisted of a PDP/11 with an RT-11 environment [10]. This is replaced by a much more powerful VME based 68020 CPU and a VxWorks environment. The new swipe survey software (*Swipe Server*) is a C++ application which creates and manipulates a *GRobot* and *JR3ForceSensor* object to monitor a JR3 force sensor mounted on the wrist of the gantry robot and make real-time trajectory corrections to the robot arm. The corrections are based on the contact force detected between the robot's end effector (swipe planchet) and the cask. The control structure maintains a  $4.0 \pm 1.0$  pound normal contact force during the swiping motion.

Figure 7 illustrates the objects that are created by the *Swipe Server*. If this figure is compared with Figure 6, it can be seen that the object hierarchies are nearly identical. The primary differences reside at the communication and servo level, where additional state machine firmware is utilized on the VME serial I/O card to handle the JR3 packet protocol and to synchronize position updates with a special trajectory card in the robot controller. At the manipulator level, the interface is identical to that found in the Cask Head Work Cell application. Again, the objects behave according to how they are created. The work cell devices can be directly controlled by objects residing on the Sun workstation whenever there are no real-time requirements, or they can be controlled in real time by objects distributed across VME CPUs.

Two additional comments can be made about the *Swipe Server* application. First, it illustrates the ability of our object-oriented environment to coexist with more traditional function-oriented environments. Rather than rewrite all of the RRSAS supervisory software, which is over 15,000 lines of C code, it was only necessary to replace a handful of modules which interfaced the supervisor to the PDP/11. The new modules create a *UnixClient* object which allows the RRSAS supervisor to communicate with the new VME-based *Swipe Server* through a *VxServer* object over the Ethernet. Everything else in the supervisor remains unchanged. Second, the *Swipe Server* utilizes *GRobot* methods which illustrate an object-oriented implementation of a task-level capability that allows the supervisor to ask the robot to "swipe the surface of a designated work piece", in this case the cask. The code segments in Figure 9 show how the generic manipulator level methods of the *GRobot* class are pieced together to create this task-level function. Similar implementations could be used in other work cells with other robots to provide this highly useful capability.

### 5. Conclusions and Future Work

The two completed implementations demonstrate that the design of RIPE has resulted in modular, reusable, extensible, and portable robot system software, and therefore has increased software development productivity and reliability of robot applications. The layered object-oriented software environment reflects the physical system. This simplifies the work for robot software developers by allowing them to construct control software environments in much the same way that the hardware system developers integrate the actual physical devices into a working system. This in turn facilitates communication between hardware and software engineers during system integration. Systems can be implemented faster due to the reusability and portability of the software. Also, RIPE can be used on most commercially available computing equipment because the development is based on a standard language and off-the-shelf operating systems. RIPE's device hierarchy and its communication interfaces which are inherent to object-oriented programming contribute to the development of a standardized Robot Independent Programming Language (RIPL) which is used to program different intelligent robot systems.

The overloading of RIPE class constructors and the use of virtual functions provides for a very flexible system with an open-ended architecture having three levels of generic interfaces. An application normally will directly create objects representing specific devices and work pieces in the work cell, whereas the communication objects will be created internally, with the details of the message transmission hidden from the applications code. This provides generic interfaces that result in architecture independence. In other words, an application can be modified to run either in a single CPU environment or a distributed environment simply by changing the way it creates its objects, and consequently the way it communicates with work cell devices.

The second level of generic interfaces exists at the device level. Because there are generic device classes for robots and sensors (*Robot*, *Sensor*, *ForceSensor*), whose attributes and member functions are inherited by specific robot and sensor classes (*CMRobot*, *LORDForceSensor*) derived from the generic classes, the application interface to a device will look the same no matter what device is used. In other words, if the Lord force sensor is replaced in the Cask Head Work Cell application by a JR3 force sensor, the application remains unchanged except for the way it creates its force sensor object (one line of code). This, in turn, leads to the third level of generic interfaces, the user level. Through inheritance and polymorphism, the same messages are sent to a *CMRobot* as are sent to a *GRobot*. Likewise, the same messages are sent to a *LORDForceSensor* as are sent to a *JR3ForceSensor*. This is illustrated by our two different work cell examples.

As a result, RIPL begins to develop as a natural consequence of RIPE. The code sequence already discussed in Figure 8 contains member function calls such as *receive\_msg* for communications, *perform* and *move\_rel* for robot control, and *set\_bias*,



*set\_output\_mode*, and *take\_reading* for force sensor control. These calls, as well as the *Robot* declarations in Figure 4, form the basis for RIPL. Currently, RIPL is an intermediate manipulator level language, upon which a task-level language is being constructed. An example of this is the swipe command used in the Radiation Survey Work Cell.

We are currently enhancing the class hierarchies for the Radiation Survey Work Cell to perform new tasks such as mating a storage cask to a storage facility door, manipulating impact limiters, and securing tiedowns. In addition, we are implementing the RIPL primitives for GMF and PUMA robot classes which will be used in future glovebox and inspection applications. We acknowledge that RIPE and RIPL must be evolutionary to be successful.

### Acknowledgments

The authors wish to acknowledge the contributions of the following: W. Davidson for his communications software and system integration support, B. Petterson for robot force control, and M. Gruesomely, C. Selleck and J. Werner for their helpful discussions in the course of this work.

### References

- [1] Backes, P., S. Hayati, V. Hayward, and K. Tso, "The KALI Multi-arm Robot Programming and Control Environment," *Proc. of NASA Conf. on Space Telerobotics*, January 31-February 2, 1989.
- [2] Blaha, J.R., J.P. Lamoureux, and K.E. McKee, "Higher Order Languages for Robots," MTIAC Report AD-A193 796, October 1986.
- [3] Cincinnati Milacron, *Communications Manual Ver. 4.0 Robot Control*, Part No.5010321-134, Cincinnati, OH 45209.
- [4] Griesmeyer, J.M., W.D. Drotning, A.K. Morimoto, and P.C. Bennett, "Cask System Design Guidance for Robotic Handling," SAND89-2444, (in preparation).
- [5] Griesmeyer, J.M., "Generalized simulation environment for factory systems," *Tools for the Simulation Profession 1989*, 20-28, 1989.
- [6] Hayward, V., and Paul, R., "Robot Manipulator Control under UNIX RCCL: A Robot Control 'C' Library," *Int. J. of Robotics Research* 5(4):94-111, Winter 1986.
- [7] LaLonde, W.R., D.A. Thomas, and K. Johnson, "Smalltalk As a Programming Language for Robotics?," *Proc. 1987 IEEE Int. Conf. on Robotics and Automation*, 1456-1461, March 31-April 3 1987.
- [8] Linton, M.A., J.M. Vlissides, and P.R. Calder, "Composing User Interfaces with InterViews," *Computer*, 8-22, February 1989.
- [9] Meyer, B., *Object-oriented Software Construction*, Prentice Hall, 1988.
- [10] Miller, D.J., "Supervisory Control for a Complex Robotic System," *Robots 12/Vision '88*, June 6-9 1988.
- [11] Nackman, L.R., M.A. Lavin, R.H. Taylor, W.C. Dietrich, and D.D. Grossman, "AML/X: A Programming Language for Design and Manufacturing," *Proc. 1986 Fall Joint Computer Conf.*, 145-59, November 2-6 1986.
- [12] Petterson, B., R. Robinett, and C. Lennox, "Lag-Stabilized Force Feedback Damping," in preparation.
- [13] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley Pub. Co., 1987.
- [14] Sun Microsystems, *SunView Programmer's Guide*, Part No. 800-1345-10, Mountain View, CA 94043.
- [15] UNIMATION Inc., *User's Guide to VAL-II, Programming Manual*, Version 2.0, #398AGI, Shelter Rock Lane, Danbury, CT. 068100.
- [16] Van Brussel, H., D. DeWinter, P. Valckenaers, and H. Claus, "A Universal Programming Structure for Multi-robot Assemble Systems," *Proc 8th Int. Conf. Assembly Automation*, 209-226, March 1987.
- [17] Volz, R.A., and T.N. Mudge, "Robots Are (Nothing More Than) Abstract Data Types," *Robotics Research: The Next Five Years and Beyond*, August 1984.
- [18] Volz, R.A. "Report of the Robot Programming Language Working Group: NATO Workshop on Robot Programming Languages," *IEEE J. of Robotics and Automation*, 4:1, February 1988.
- [19] Wind River Systems, Inc., *VxWorks Reference Manual, Ver 4.0*, Emeryville, CA 94608.



# SIC — a System for Stochastic Simulation in C++

Bernd T. Kluth

Institute for Teleprocessing  
Aachen University of Technology  
Aachen, FRG

E-mail: bernd@dfv.e-technik.rwth-aachen.de

## Abstract

This paper describes a system for supporting stochastic discrete event simulation. Its support of simulation concerns three fields: First, a library of simulation tools for random number generation and statistical analysis is offered along with a mechanism for implementing process oriented simulation. Second, a modeling toolkit has been set up and may easily be extended by the user, facilitating model construction and reuse of components in different programs. Third, based on such a toolkit and the SIC program structure, a system for graphical generation of simulation programs has been established with an X windows based graphic editor. The reusability of model components offered by the object oriented design techniques of C++ are illustrated by examples of simulation models.

## 1 Simulation Issues

Discrete Event Simulation is still the most widely used means for analyzing the performance of communication and other randomly driven, complex digital systems. The reliability of the results obtained by simulation massively depends on the statistical qualities of the random number generators applied and on the methods used for analyzing the data observed during a simulation run. Their accuracy of course rises with the number of observations made. This leads to a demand for high event counts, which in turn calls for a good efficiency of the program execution. Finally, the methods used for mapping a model into a simulation program influence the modeling safety.

The significance of the latter aspect increases as the investigated models grow more complex. In the context of simple models the main issue was to retain the model's *functionality* inside the simulation program. To maintain models of higher complexity more safely, it is necessary to retain the model's *structure* as clearly as possible in a simulation program. But even smaller programs of course may profit by a good program design.

## 2 Object Oriented Simulation

Recently, one topic seems to dominate many new software systems: object orientation. While the basic idea was originally established in the context of simulation programming by the language SIMULA, its recent upshot originated from other projects such as

SMALLTALK. The use of these systems in simulation programming, however, is impeded by their relatively poor performance, as compared to "traditional" programming languages. On the other hand, it has been noted before [11] that object oriented concepts can help to maintain the model's structures inside a simulation program in a natural way. The rationale for this is that simulation models usually describe the interactions between different entities inside a system. This is naturally reflected by object oriented structures. Using object oriented techniques may thus advantageously influence the reliability of the resulting model, for the correctness of the program can be assured more easily and thus more safely. As is well known, this perception also was a motivation to develop the language C++ [9], adding object oriented programming concepts to the language C. When introducing these new language features, care was taken to maintain as much efficiency as possible. Thus, C++ obviously combines the expressive power of object orientation with the efficiency of the C programming language making it ideally suited for simulation programming.

### 3 SIC

Written entirely in C++, a simulation system called SIC<sup>1</sup> was developed. A portable kernel enabling process oriented simulation is at its core. Some of its concepts are influenced by [10], differences exist however in the way how process oriented simulation and essential support functions are implemented.

The key aspects of SIC are presented in this section. Based on the syntax of C++, SIC gives a framework for defining a process as well as for setting up an entire simulation program. On top of that, a set of building blocks for creating queueing network simulation models have been set up allowing simulation programs to be established in a well structured manner.

#### 3.1 SIC Process Structure

Processes are defined inside SIC as classes. A process class is derived from a base class *task*. The definition of a process consists of three distinct and distinguishable parts which naturally match the modeler's view:

##### **persistent data:**

The persistent data comprise those variables of a process which describe its state and which thus have to retain their values during deactivation phases of a process. They are contained in the class definition of a process class.

##### **initialization:**

The initialization of a process serves to maintain correct initial values of a process's persistent data. Additionally, it serves to *parameterize* a process. This initialization is performed by the constructor of the process class.

##### **algorithmic behavior:**

The algorithmic behavior of a process is specified in a special member function of a process class called *run*. This function defines the operation of a process, its cooperation with other processes and its access to resources.

---

<sup>1</sup>SIC: Simulation in C++

Processes the active entities inside a simulation model. In stochastic simulation, processes usually execute repetitively while interacting with their neighbors. Such interactions, however, may require the execution of a process to be suspended either unconditionally or until some conditions are met, e.g. until some resources are freed. Later, they have to be resumed at the point of their previous interruption. This leads to a behavior of processes which is usually referred to as co-routine execution.

As C++ itself does not offer co-routine programming, this has to be introduced artificially to allow for process oriented simulation. One possible solution is offered by the task library supplied with AT&T C++ Version 1.2.1. This solution has the advantage of allowing a process change at any level of an arbitrary hierarchy of procedure calls. It was ruled out, however, for the sake of portability.

Process oriented simulation is thus implemented in SIC by mapping the co-routine behavior of a process to procedure calls. A preprocessor ensures that execution is resumed at the appropriate point in the sequence of a process's actions. This implementation does not depend on additional routines in assembly language. Due to this, SIC meanwhile was easily installed on various UNIX systems. An aspect of this method is that statements which (possibly) interrupt processes may only be placed in their run member function and not in an arbitrary function called from run (although it is of course possible to call other functions from the run function). So far, this restriction has not been an impediment for the implementation of simulation models, and from the experience gained the ease of porting seems to have outweighed that aspect.

As mentioned, processes are defined inside SIC as classes. By setting up a process's class description, only the *template* of a process is given; the *incarnation* of a process is carried out separately. The term *incarnation* is used in this paper to denote both the action of creating an object from a C++ class and the resulting object itself. This concept allows for several process incarnations to be created from one process definition. As it is possible to parameterize a process in SIC, each process incarnation may be set up individually, e.g. concerning its stochastic characteristics and the model resources it utilizes. Thus sets of similar, though slightly differing processes may be maintained efficiently in SIC.

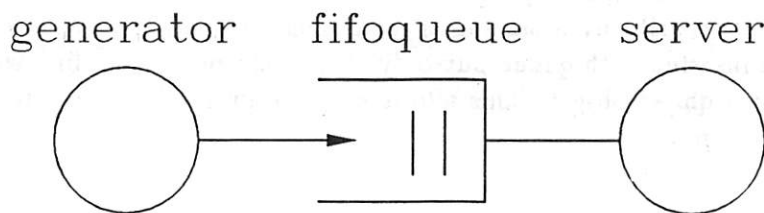


Figure 1 : Process model of a simple queueing system

As an example, the model of the simple queueing system shown in Fig. 1 will be considered. The class definition of its generator process is given in Fig. 2. The constructor's argument list indicates that the queue into which the generator process writes messages is passed to the process at incarnation time by `qtail`. The same applies to the random number generator defining its stochastic behavior. In its member function `run`, which is not shown here, the generator creates a message object and marks it with the current simulation time before inserting it into the `qtail`.

```

class generator : public task {

    //persistent data of a generator process; pointers to:
    qtail *      tail;          // queue written to
    basegen *    rgen;          // random number generator
    queue_object* obptr;        // a message object

    void    run();              // called by the scheduler

public:
    generator( char *, qtail *, basegen * );
};

```

Figure 2 : Class definition for the generator process in Fig. 1

### 3.2 SIC Program Structure

Resulting from this process structure, a typical SIC program looks as shown in Fig. 3. Its first part consists of definitions of processes in the above mentioned form and of passive components used to "connect" the processes with each other. It is not till the subsequent main program that incarnations of these components are *created* and parameterized. In the context of queueing models, the structure of the investigated system is thus defined in the main program, specifying both the model's topology and parameterization.

In Fig. 1, for example, the processes *generator* and *server* cooperate via a fifo queue object. This is a specialization of a general queue object in SIC which forwards messages in FIFO- (First-In-First-Out-) order, that is the same order as they were inserted. Other queueing disciplines are possible as well. Generally, queues in SIC appear to the user as two different classes — *qhead* and *qtail*, which are interconnected internally. This concept improves modeling security by assigning specific operations to a queue's head resp. tail: messages may be inserted with *queue\_put* only at a *qtail* object, reading with *queue\_get* is only allowed at a *qhead* object. This allows some programming errors to be discovered already at compile time.

### 3.3 A Modeling Toolkit

Process specifications and the main program may of course be distributed to different modules which are translated separately. This offers a path to elegantly establish a library with components of simulation models. Their use considerably reduces the effort of writing a simulation program while simultaneously advancing its clearness.

A toolkit of components for queueing network models was established for SIC. It contains several predefined process types for setting up the active parts of a simulation program. Additionally, building blocks for modeling the interaction of processes are included. These are various queue objects according to the structure mentioned above, and semaphore objects for controlling the access to resources.



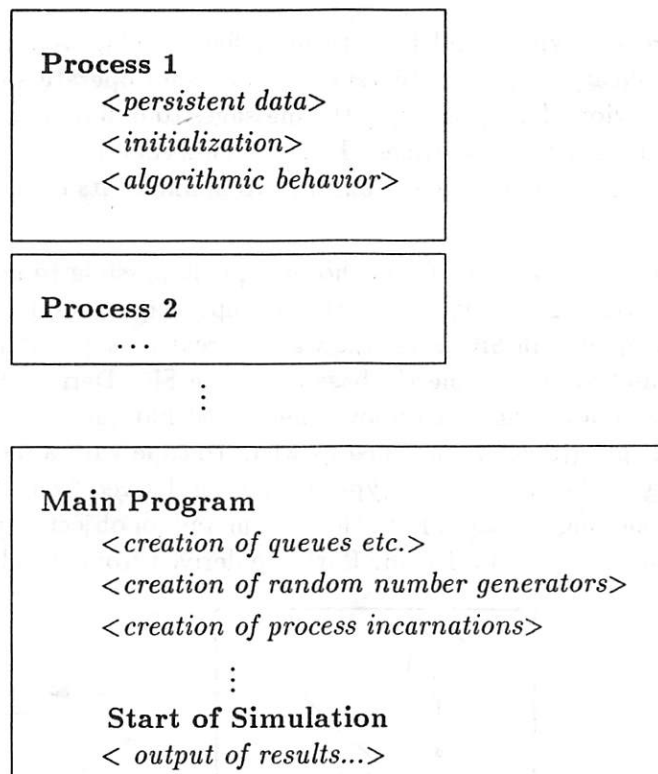


Figure 3 : Structure of a SIC program

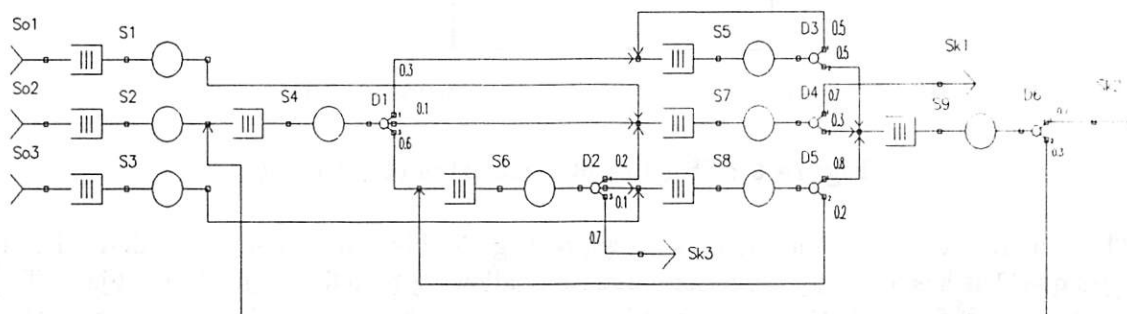


Figure 4 : Queueing model with several stations

Another advantage of distinguishing between head- and tail-objects of queues becomes apparent when considering the open queueing network model in Fig. 4. This model is taken from an article on approximate analysis of queueing networks [6], it consists of nine stations S1,..S9. Each station in turn consists of a server, drawn as a circle here, and a queue. All servers operate in the same manner: they pick a message from a queue's head, process this message, and forward it into another destination after simulating their service time. Thus it seems obvious to create all server incarnations from the same process class. Taking a closer look, however, one notes that some servers put their messages immediately into queue-tails, some lead to exits from the network Sk1,..Sk3, others forward them through branching points D1,..D6 into one of several possible directions with a specific probability assigned to each branch. If implemented in a straightforward way, for each of these cases

a specific server process type would have to be defined. This would make the resulting program more complicated. As the different servers would operate similarly with respect to their internal behavior while forwarding the messages to different targets, the same code would have to be replicated several times. This prevents code reuse and rises the need not only to ensure its correctness repeatedly, but also to maintain its equivalence after changes have been applied.

Using the object oriented features of C++, however, it is possible to develop this model in a more structured way. A generalization of the concept of queues introduced by setting up a class hierarchy for queues in SIC paves the way to treat all servers in this model equally. The classes `qhead` and `qtail` are merely base classes in SIC. Derived from these there are several queue types, among them the above-mentioned `fifo` queue. Queues implementing other strategies, e.g. priorities, are included as well. To cope with a model where messages are routed as in Fig. 4, two new object types have been derived from the base class `qtail`: a `sink` object for removing messages from the system and an object implementing a branch for the flow of messages called `decision`. Both are derived from the class `qtail`.

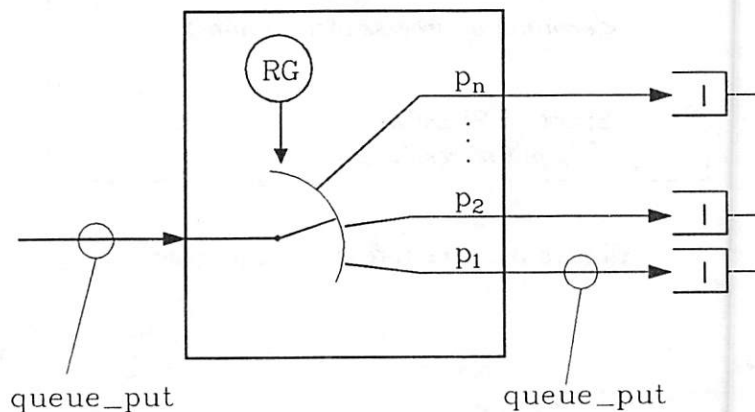
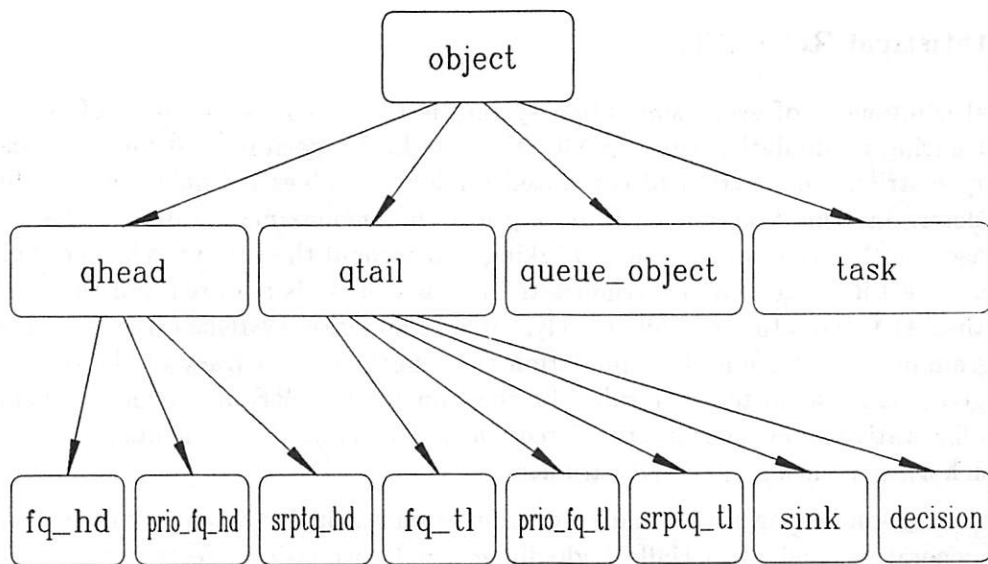


Figure 5 : Structure of a decision object in SIC

The structure of a decision object is shown in Fig. 5. Like any other object derived from class `qtail` it has a member function `queue_put` allowing to deliver a message object. This function itself forwards the message object to one of a choice of `qtails`, again using their `queue_put` member function. Internally it uses a random number generator `RG` to choose the target direction. Each branch therefore has a branching probability  $p_i$  assigned to it. In order to take advantage of this scheme, processes in the toolkit are written to operate on objects of the base class `qtail`. Their incarnation may thus be "connected" to a "real" queue as well as to the decision object described. Similarly, a `sink` object is available. It also has a member function `queue_put` which deletes the message objects received.

Using these components for constructing a simulation program allows each server process in the model Fig. 4 to be written to forward its messages into an object of `qtail` type. This allows for all process incarnations to behave identically and thus to be created from the same process class. Only one additional process type has to be defined to implement the stations `So1`,...`So3`, where messages are generated and entered into the network. Thus the use of inheritance helps to reduce program size and to keep the program more comprehensible because processes with the same internal behavior have to be maintained only once.



**Figure 6 :** Excerpt of the SIC class hierarchy

An excerpt from the resulting class hierarchy in SIC is shown in **Fig. 6**. The classes `qhead` and `qtail` are merely base classes for queues. They are themselves derived from a common base class `object`, together with the class `task` for processes and `queue_object` for message objects. Derived from `qhead` resp. `qtail` are classes for FIFO-queues (`fq_hd`, `fq_tl`), classes implementing queues for non-preemptive priorities (`prio_fq_hd`, `prio_fq_tl`) and for the SRPT<sup>2</sup> strategy (`srptq_hd`, `srptq_tl`). SRPT is a preemptive scheduling procedure where the processing of a message is suspended whenever a shorter one enters the queue. It has been shown that the SRPT strategy minimizes sojourn times in a waiting room and is optimal in this respect over any other scheduling strategy [7], even if taking into account the overhead times induced [2]. By including SRPT scheduling in the SIC toolkit, its effect on sojourn times in the complex network in **Fig. 4** could easily be examined [5]. Here another advantage of the toolkit approach can be noted: it allows the implementation of the scheduling strategy to be verified in a simple model whose results can be obtained analytically. Without touching the code it can later be integrated into the more complex model. **Fig. 6** also shows the classes `sink` and `decision` being derived from class `codeqtail`.

Structuring a simulation program in the above-mentioned fashion considerably improves its maintainability. Because its components are set up schematically, changes in the model's behavior due to the modification of single components of the model may easily be examined by merely manipulating these components, leaving the rest of the program untouched. Variations of queues for instance may easily be introduced, where messages instead of the usual First-In-First-Out- (FIFO-) order are forwarded in Last-In-First-Out- (LIFO-) order or according to some priorities. Including them in a SIC simulation model only requires to replace the queue objects used so far. The code for the processes communicating via these queues may be left untouched.

<sup>2</sup>SRPT: Shortest Remaining Processing Time first

### 3.4 Statistical Reliability

A central component of every simulation system is the statistical analysis of the values observed during a simulation run. In SIC, the new LRE algorithm<sup>3</sup> [8] for obtaining the stationary distribution function of correlated random variables is implemented, which is able to determine this function with an a priori chosen maximum value of the relative error. Based on the accuracy desired and taking into account the variable's local correlation coefficient, the LRE algorithm determines the number of trials required and thus controls the length of the simulation run objectively. Other simulation systems often leave it up to the programmer to determine the simulation run length; the accuracy of the observation is then given as an a posteriori result. In addition to the LRE algorithm, conventional methods for statistical evaluation are offered which determine the moments of observation values such as their mean value and variance.

Another factor influencing the reliability of simulation results is the quality of the random number generators used. Especially high-dimensional correlation effects of these random numbers are among the less obvious factors which influence the reliability of simulation results. Because of this, SIC offers a choice of random number generators. Besides conventional pseudo random number generators, a quasi-ideal table random number generator based on physical phenomena is implemented [3].

## 4 Graphical Program Generation

Program generators are often used to simplify the creation of programs in a special and limited context of applications. They usually compose a program by arranging and parameterizing predefined components according to user specifications. For the sake of efficiency and simplicity, they often immediately generate machine code or any other intermediate language which the average user typically is not acquainted with.

The SIC program structure naturally leads to a separation between the definition of the model's components and their incarnation and parameterization. This not only enables one to establish a modeling toolkit in form of a library of simulation model components. It also forms a basis to further facilitate the generation of simulation models by program generators. In conjunction with a library of model components as described in section 3.3, a program generator may concentrate on creating the main program for SIC defining the model's structure and parameterization.

A system for graphical generation of queueing network simulation programs was developed for SIC. It is based upon a graphic editor written in C++ called NETCAD running under the X window system. The program generator integrated into this graphic editor is able to create a SIC program from a schematic drawing that the user has composed. An example for its operation is the network diagram in Fig. 4. It was created using the graphic editor NETCAD. After interactively adding parameters to the model, a SIC program may be generated from this drawing.

For drawing simulation models, a multi window mode has been added to the graphic editor. In this mode, diverse model components are offered in a menu which may be used to compose a queueing network model. A view of the editor screen is given in Fig. 7. After

---

<sup>3</sup>LRE: Limited Relative Error



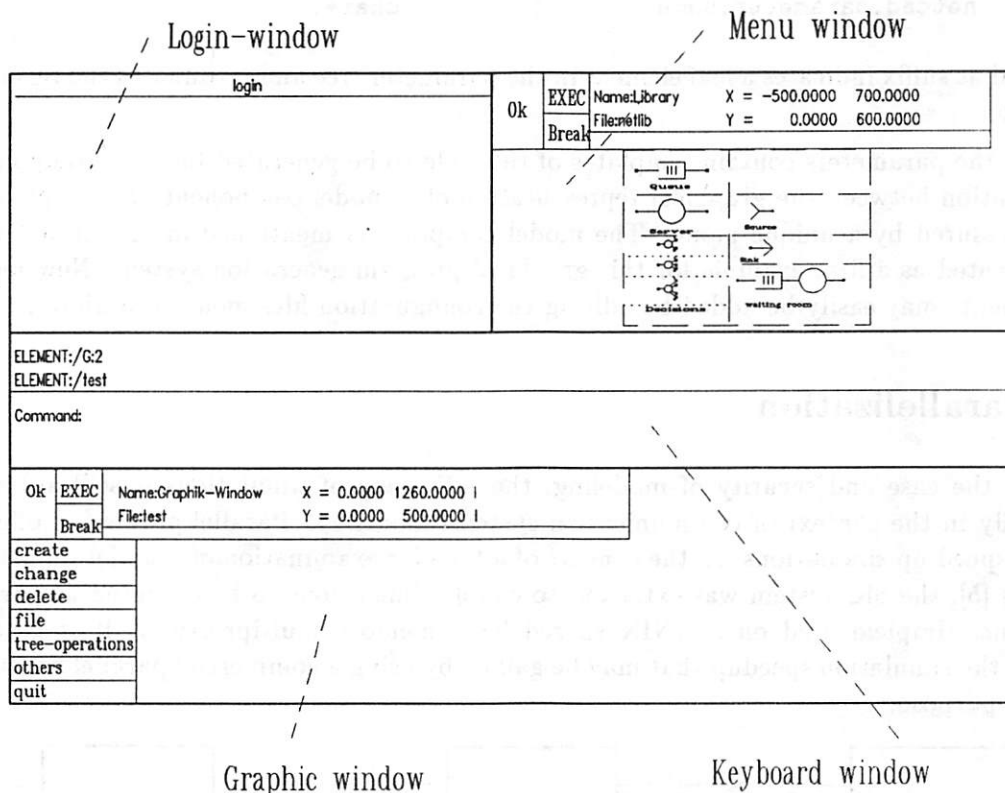


Figure 7 : Screen view of the NETCAD graphic editor

parameters have been added, a simulation program may be generated, translated and run without leaving the editor. The results of such a simulation run may in turn be visualized in the editor display, e.g. in form of distribution functions.

The program generator creates programs which look like those written manually. This enables the user to quickly create a program source by means of the graphic editor which later on he may extend and modify manually, if desired. It also increases transparency for the user enabling him to read and understand the code produced by the generator.

When designing this program generator system, care was taken to ensure that the introduction of new model components is achieved by merely setting up configuration files. These are comprised of graphic files which contain the graphical appearance of model components and parameter files which describe their logical structure. As an example for the structure of the parameter file, the generator process shown in Fig. 2 is considered. A line in this file specifies the parameters required to define this process.

```
netcad.parameter.generator      name;rng|
```

Built according to the X windows conventions, this line states that a name may optionally be assigned to the generator. It is required to state a random number generator (rng). Whether a parameter is optional or required is controlled by its suffix (here ';' resp. '|') in the definition file. The meaning of the variables on the right side of a definition is in turn defined recursively:

A period as suffix indicates a leaf element in the parameter tree and terminates the recursive definition.

Finally, the parameters contain templates of the code to be generated for each component. The relation between the graphical representation of a model component and its parameters is assured by a unique name. The model components mentioned in section 3.3 were implemented as a first example for this graphical program generation system. New model components may easily be added by editing the configuration files mentioned above.

## 5 Parallelization

Besides the ease and security of modeling, the *efficiency* of simulations is still an issue, especially in the context of communication systems modeling. Parallel processing offers a way to speed up simulations. In the context of a broader examination of parallel simulation systems [5], the SIC system was extended to enable simulations to be executed as parallel programs. Implemented on a UNIX shared local memory multiprocessor, it is used to explore the simulation speedup that may be gained by using a commercial parallel hardware for this purpose.

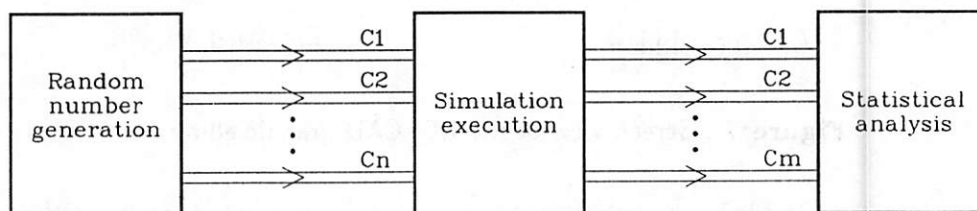


Figure 8 : Functional parallelization in SIC

The functional parallelization approach is used to parallelize SIC programs. In this approach, certain functions of simulation programs are distributed to several processors. This kind of task division may be performed in a way independent of the model examined. In the parallel version of SIC, the generation of random numbers and the statistical analysis are carried out by separate processors respectively (Fig. 8). This approach was also investigated on a special hardware [1], [4]. The tasks distributed here can work independently from each other, e.g. random numbers may be generated in advance and simulation results may be analyzed asynchronously from their production. A set of buffered communication channels  $C_1, \dots, C_n$  is required for their cooperation, providing for transfer of the different types of random numbers used in the simulation model, or of the results obtained at different observation points in the model respectively.

Using the object oriented features of C++ and based upon the structure of SIC it is possible to translate the same simulation program source to either a sequential simulation program or to a parallel one. The internal structure of SIC forms a basis to achieve this in a way transparent to the user: both random number generators and the statistical analysis are called in SIC via member functions of their respective classes. In the single-program case, these classes are implemented to perform the desired operations themselves. In the parallelized program, they act as interfaces to the communication channels mentioned above. Thus the parallel

version is integrated mainly by providing an alternate library for some internal classes inside SIC and by a preprocessor, which serves to create the additional processes needed.

## 6 Conclusion

In the context of SIC, C++ has proven well suited for simulation programming in several respects. Besides the obvious fact that its object oriented features enable the implementation of a model scenario in a natural way, it maintains an efficiency of program execution which allows the utilization of these features adequately in a simulation without paying a performance penalty. The separation between the definition of model components and their incarnation, which facilitated to set up the graphic program generation system, was well supported by C++. C++ also helped to implement two versions of SIC offering transparent parallelization. As version 2.0 of the C++ translator was not yet available when developing SIC, multiple inheritance could not be used. This required the use of `friend`-attributes in places where a cleaner relation would be desirable.

An extension that is planned for SIC concerns the event list. Depending on its size, different algorithms are optimal for its implementation. Choosing the appropriate algorithm respectively may avoid wasting resources. Therefore, a dedicated event set class will be introduced for the implementation of this feature. Further, it is planned to include provisions for animated simulation into the simulation system resp. the graphic editor. Such a feature has educational as well as practical aspects: while helping to understand *how* simulation works in general, it may also help to verify the correct operation of a simulation program interactively.

## 7 Acknowledgement

I wish to thank my colleagues, especially Dr. Carmelita Görg, who helped me by revising this paper and by various discussions. Further I'd like to thank Prof. Dr.-Ing. F. Schreiber for the continuing support of the project, inside which the SIC system was developed, and for discussions on this paper.

## References

- [1] Barel, M.: *Performance Evaluation of the Discrete Event Simulation Computer DESC*. Proceedings 18th Annual Simulation Symposium Tampa 1985, pp. 11-13.
- [2] Görg, C. *Evaluation of the Optimal SRPT Strategy with Overhead*. IEEE Trans. Comm. (1986), Vol. COM-34 No. 4, pp. 338-344.
- [3] Gude, M. *Concept for a High Performance Random Number Generator Based on Physical Random Phenomena*. FREQUENZ 39 (1987) 7/8.
- [4] Kluth, B.; Görg, C.: *Performance Evaluation of a Cost Effective Multiprocessor for Teletraffic Simulation*. in: *Teletraffic Science for New Cost-Effective Systems, Networks and Services*. Proceedings 12th International Teletraffic Congress; Torino, Jun 1988, ed. M. Bonatti, North Holland

- [5] Kluth, B.: *Multiprocessor architectures with functional parallelization for stochastic simulations*. PhD. Dissertation, Aachen University of Technology 1990.
- [6] Kühn, P.J.: *Approximate Analysis of General Queueing Networks by Decomposition*. IEEE Trans. Comm. 1979, pp.113-126.
- [7] Schrage, L.E.; Miller, L.W.: *The Queue M/G/1 with the Shortest Remaining Processing Time Discipline*. Operations Research 14 (1966) 670-684.
- [8] Schreiber, F.: *Effective Control of Simulation Runs by a new Evaluation Algorithm for Correlated Random Sequences*. AEU 42 (1988) pp. 347-354.
- [9] Stroustrup, B.: *The C++ Programming Language*. Addison Wesley 1986.
- [10] Stroustrup, B.: *A Set of C++ Classes for Co-Routine Style Programming*. AT&T Bell Labs. Computer Science Technical Report.
- [11] Zeigler, B. P.: *Hierarchical, Modular Discrete-Event Modeling in an Object-Oriented Environment*. SIMULATION 49 (1987), pp. 219-230.



# A Type Parameterization Language for C++

Richard Blinne

NCR Microelectronic Products Division  
Fort Collins, Colorado

## Abstract

Several schemes have been presented in the past to add type parameterization to C++. Parameterization allows software developers to build re-usable data types that are *type-safe*. In these systems the parameters are, for the most part, syntactic constants. We have found there are certain situations where it is convenient to have meta-variables that can be used in addition to, or in lieu of, these constant syntactic parameters. In response to this we wrote a small, special-purpose interpreter that reads a parameterization language; this interpreter is used to instantiate type-free code templates into C++ code that is type-specific.

This language supports:

1. Assignment and other operations on meta-variables.
2. Normalization of the indirection level of C++ variables whose indirection level is unknown until compile time.
3. Conditional Expressions.
4. Regular expression comparison.
5. Boolean variables for conditional compilation.
6. Differentiation of C++ primitive types from (user-defined) composite types.

This paper describes this parameterization language, the interpreter, and our experience building container classes from such a language. We will also discuss how our methodology might be integrated into Bjarne Stroustrup's proposed additions to the C++ language.[1, 2]

## 1 Introduction

The ability to do type-safe "container classes" in C++ has been a major goal for several years. So far, there have been two approaches to reach this goal. They are inheritance from a void\* base class and some type of parameterization scheme. Since type parameterization schemes can generate type-safe interfaces to void\* classes the latter approach can subsume the former.[1]

There have been at least three approaches to type parameterization. They are:

1. Generic classes built by C preprocessor macros.
2. Prototype macroization using sed and other text replacement utilities.
3. Building class templates into the C++ language.

The first approach is used in the `generic.h` package provided with the current implementations of C++.[3] While this can build "generic" classes, building class templates remains a cumbersome process.

The second approach, which is used within `libg++`, makes it easier to write templates than the first approach.[4]

The last approach, which is proposed by Stroustrup as an experimental extension to C++, appears to be the most natural way to add type parameterization to the C++ language.[1] Obviously, this changes the definition of C++.

Our approach to the problem was to build a special purpose text-replacement utility in conjunction with an interpreted control file. The following sections will describe this approach.

Our *parameterization program* reads a *template file(s)* and a *control file* and produces an *instantiated file(s)*. All the following examples will identify the file type from which it came. The type of file will be identified with `typewriter` type. The parameterization program will also be referred to as the *interpreter*.

Commands for the *control file* will be in **bold type**. Meta-variables (and their instantiations) will be in *sans serif type*.

## 2 Program Architecture

The parameterization program performs the following steps to convert type-free code into type-specific code:

1. Find and open the **control file**. The **control file** is found via a search path mechanism. This mechanism is similar to the search path used by shells to find commands.
2. Interpret the **control file**. The following steps are the result of interpreting the control file.
3. Bind meta-variables.
4. Read the appropriate **template files**.
5. Write the appropriate **instantiated files** using the bound values of the meta-variables.

It is the **control file** which makes our approach unique. The existence of the control file allows us to conditionally build different implementations of our type-free classes. The **control file** also better documents our type-free classes. Users are asked questions about their type-specific classes. Extended comments can be made to the user explaining the assumptions and the limitations of the parameterized classes.

### 3 Meta-Variables

*Meta-variables* are variables that are bound before run-time. In our approach, meta-variables are bound when the **control file** is interpreted. In Stroustrup's approach, meta-variables are bound at compile-time.

The most common meta-variable is the *type variable*. Type variables are bound to type names.

Meta-variables appear in the **template file** surrounded by chevrons (<>). This makes them easy to find in a text replacement tool (or compiler). The interpreter via the **control file** establishes the values of these meta-variables. Once these variables are bound by the interpreter, they are applied to the **template file(s)** and written out into **instantiated files**.

#### 3.1 Binding Meta-Variables in the Control File

Meta-variables are bound either by querying the user with the **Ask** statement, or by assignment using the **Set** statement.

The following is an example segment from a **control file**:

```
Prompt "What is your type?"
Ask MyType[Actual|Pointer]
Set ClassName[Actual] = $MyType | list
```

The session it would produce would look like:

```
What is your type? String
```

In this example, we prompted the user for a type. **ClassName** will be the string concatenation of what was entered with "list". In this example, **MyType** would be bound to **String** and **ClassName** would be bound to **Stringlist**.

#### 3.2 Type Meta-Variables

There are three kinds of *type meta-variables*, *Actual*, *Pointer*, and *Reference*.

An *actual* type variable has no asterisks or ampersands in its definition. A *pointer* type variable has one or more asterisks in its definition. This means that

*pointer* variables can have more than one level of indirection. The interpreter keeps track of this when it is instantiating these variables. A *reference* type variable has an ampersand in its definition. A type meta-variable can be both a *pointer* and *reference* meta-variable. When variables are queried, the system checks to make sure type variables conform to these definitions.

Refer to the previous example. In this case, `MyType` may be either an *actual* or a *pointer* type. This means our template definition can handle either kind of type variable. This mechanism allows the template developer to make sure the types used in his template are well-formed.

### 3.3 Instantiating Meta-Variables

Meta-variables are instantiated from the `template file` to the `instantiated file`. *Instantiating* is replacing the reference to the meta-variable in the `template file` with the value bound to it by the `control file`. This is called *normal instantiation*.

### 3.4 Instantiating Type Meta-Variables

A type meta-variable can be instantiated in one of three ways. These three ways are:

1. Normal Instantiation
2. Converted Instantiation
3. Cast Instantiation

#### 3.4.1 Normal Instantiation

A *normal* instantiation replaces the type variable by its bound value. For example for the `template file`:

```
<Type> MyVariable;
```

If `Type` was bound to `String*`, this would become in the `instantiated file`:

```
String* MyVariable;
```

#### 3.4.2 Converted Instantiation

*Converted* instantiations take the root of the type description and add the appropriate characters to make them pointers, references, or actuals depending on the syntax. For example the `template file`:



```

<Type*> MyVariable1;
<Type&> MyVariable2;
<Type:r> MyVariable3;

```

If Type was again bound to String\*, would look like this in the instantiated file:

```

String* MyVariable1;
String& MyVariable2;
String MyVariable3;

```

As shown in this example, <Type\*> converts Type to a pointer type with one indirection level, <Type&> converts Type to a reference type, and <Type:r> converts Type to its "root" (no indirection) type.

### 3.4.3 Cast Instantiation

The *cast* instantiation allows an arbitrary type to be "cast" into a known level of indirection. This is also called *indirection level normalization*. For example, the template file:

```

<Type1*> MyVariable1 = <(Type1*)> YourVariable1;
<Type2:r> MyVariable2 = <(Type2:r)> YourVariable2;

```

If Type1 was bound to String and Type2 was bound to String\* would produce the following instantiated file:

```

String* MyVariable1 = & YourVariable1;
String MyVariable2 = * YourVariable2;

```

The syntax and the name cast initialization was chosen because it is similar to casting between variable types. But in this case we are casting between indirection levels.

## 3.5 Literal Meta-Variables

*Literal* meta-variables are bound to simple character strings. Literal meta-variables are not checked to see if they are actuals, pointers, or references. When instantiated, literal meta-variables expand into their bound values.

Literals have many different uses. They could be used to enter code fragments. For example, for the template file:

```

if (<Compare>)
{
    // swap elements
}

```

the user can enter his own comparison test.

In this case, we don't need to have a convention that defines what a comparison function looks like. It need not be a < operator nor a function with a set name. The developer could just enter in the code fragment that does the compare.

This can be done in other ways, but we found that the interpreter was asking too many questions. With the literal variable, we just let the user enter in the appropriate information.

Another use of literal functions, is when a class may or may not be used in any further inheritance. The virtual keyword could become a literal meta-variable. The control file would look like:

```
Prompt "Do you want to inherit from this class?"
Ask IsInherited[boolean]
if IsInherited
    IsVirtual[literal] = virtual
else
    IsVirtual[literal] = ""
endif
```

The corresponding template file would look like:

```
class MyClass
{
    public:

        <IsVirtual> void MyFunction();
}
```

With this scheme, if we don't need inheritance then we don't pay the overhead.

### 3.6 Boolean Meta-Variables

*Boolean* meta-variables are used in conditional expressions (described in the next section) and to conditionally "comment out" sections of code. There are two types of Boolean variables. The first is a normal Boolean variable created by *Ask* and *Set*. The second is a *primitive* type variable created for each type variable that exists.

The primitive type variable is true if the defining type is a C++ primitive (e.g. `int`, `char`, `char*`) and false if it is not. The test for the primitive type variable is the type variable name with `:p` appended to it. This type of Boolean variable can be used to determine whether to pass a variable into a function as a reference.

For example, the control file would look like:

```

if Type:p
    Set ElementPassed[Actual] = $Element
else
    Set ElementPassed[Reference] = "const " | $Element | &
endif

```

and the corresponding template file would look like:

```

void MyFunction(<ElementPassed> element);

```

Boolean variables are also used to comment out sections of code. For example, if a class uses `void*` then the ostream operator for that class is inappropriate. This is because the ostream operator does not handle `void*` in an intelligible fashion. This is what the control file would look like:

```

if Element == "[ ]*void[ ]*\*[ ]*$"
    Set VOID_ELEMENT[Boolean] = yes
    Prompt ""
    Prompt "The ostream operator will be commented out because"
    Prompt "you declared the type of the element as void *."
else
    Set VOID_ELEMENT[Boolean] = no
endif

```

The corresponding template file would look like this:

```

<BEGIN !VOID_ELEMENT>
#include <stream.h>
<END !VOID_ELEMENT>

.
.
.

class MyClass
{
<BEGIN !VOID_ELEMENT>
    friend ostream& operator<< (ostream& OutputStream,
                                <MyClass&> Class);
<END !VOID_ELEMENT>
};

```

The `<BEGIN>` and `<END>` statements in the template file are used to "comment out" sections of code.

## 4 Conditional Expressions in Control Files

We have already used some conditional expressions in our previous examples. Conditional expressions are nested if/else/endif structures. What is tested inside the if are the following:

1. Boolean Meta-variables
2. Inverses of Boolean Meta-variables
3. Regular Expression Equality
4. Regular Expression Inequality

Here are examples of each in order:

```
if MyBoolean
  Prompt "MyBoolean is true."
endif

if !MyBoolean
  Prompt "MyBoolean is false."
endif

if MyVariable == "foo"
  Prompt "MyVariable matches the regular expression foo"
else
  if MyVariable != "f.o"
    Prompt "MyVariable does not match the regular expression f.o"
  endif
endif
```

## 5 Subprocesses and File I/O

Our `control file` allows the template developer to control which files are read in and written out by the system. The `control file` also allows him to create subprocesses. There are two commands to support this: the `Write` and `Do` commands, respectively.

`Write` reads in template information, parameterizes it, optionally pipes it through a filter, and writes it out to a file. The following is an example of a `Write` command:

```
Write < MyClass.hP | $MyFilter > $OutputFile
```



In this case, we parameterize the file `MyClass.hP`, filter it through the `MyFilter` command (which is a meta-variable), and output the results to the file specified by the literal meta-variable `OutputFile`.

The `Do` command creates a subprocess and optionally redirects the standard output of that process. If we wanted the user to edit one of the files we just created, we might do the following:

```
Set MyCommand[literal] = $MyEditor | " " | $OutputFile
Do $MyCommand
```

The `Write` command uses a search path to find its files. This allows template developers to place their `template files` and `control files` in a library that can be found via a search path.

## 6 Uses of the Parameterization Utility

So far, we have used our parameterization utility in three ways; we have generated full implementations, generated type-safe interfaces from base classes, and generated extensible data objects. In fact, our templates can generate any one of these styles based on the desires of the user. We will explore the pros and cons of each approach in the next sections.

### 6.1 Full Implementations

*Full implementations* generate all the methods separately for each of the different instantiations of a template. Full implementations can unnecessarily bloat our objects and generate excess code. This is because we do not reuse what is in common between the different instantiations. So, in general, we don't want to generate full implementations. However, there are cases where this is unavoidable as shown below.

### 6.2 Generating Type-Safe Inherited Classes

Most generated classes are done using *type-safe inherited classes*. A `void*` base class is first generated and put into a library. For any instantiation of the `template file`, we generate a type-safe interface that does all the casts from `void*` to the user specified type. Generating classes using our utility avoids the coding errors caused by generating type-safe interfaces by hand.

There are cases that won't work using this method, though. Many developers cannot afford the extra space taken by a pointer to an object and want to store the object directly within the data structure. Such an approach is incompatible with inheriting from `void*`. In this case, a full implementation is generated. Our system is sufficiently flexible that the developer can determine his needs and generate the appropriate implementation. He does not have to settle for a single "catch all" implementation.

### 6.3 Extensible Data Objects

Using *extensible data objects* is an extension of the type-safe interface concept. In addition to generating the type-safe interface, new data members and methods can be added to the class based on the user's needs.

We want to create "lean" data objects with only the bare essentials. Tool developers can then add what is appropriate. If the underlying templates change, the extensions can be automatically reintegrated into these classes. An example template file could look like this:

```
class <MyExtendClass:r> : public <MyClass:r>
{
    public:
        <MyExtendClass:r> MyFunction()
        {return (<MyExtendClass:r>) <MyClass:r>::MyFunction();}
#include <UserExtensions>
};
```

UserExtensions is a file name that contains all the extensions the user wants to add to the class. MyFunction is not a virtual function because the return type is changed from the base class.

## 7 Proposals for Extension to C++

The following sections are some suggestions for extensions to Stroustrup's proposed parameterization of C++[1], based on our results.

### 7.1 Binding Meta-Variables With A Control File

In addition to meta-variables being passed in as parameters, binding meta-variables with a control file is useful. We found that such an approach improves the flexibility of our templates immensely. It is easier for users to understand our templates because of the prompting involved. The conditionals in the control file allows us to have different implementations from which the user can choose.

### 7.2 Compile-time conditionals

In his paper, Stroustrup cogently argued why conditionals of run-time type variables would be difficult to implement.[1] If, on the other hand, a restricted compile-time only conditional were available, then these restrictions wouldn't apply. While this would not implement truly heterogeneous container classes, we can still get some useful results out of this feature.

Compile-time conditionals could be distinguished from their run-time counterparts by having their parameters contained by chevrons rather than parentheses. Only compile-time meta-variables could be used in these statements. If

the if or case statement passed, the code in that section would be generated otherwise it would not.

An example would look like:

```
if <Type == "int">
{
    // Code that was specific to an integer type
}
else
{
    // Other code
}
```

Compile-time conditionals are analogous to our <BEGIN> and <END> statements.

### 7.3 Type Instantiation Variables

We found that having more than one kind of instantiation of type variables was useful for dealing with elements that can be both pointers and actuals. The addition of these kinds of instantiation would also increase the flexibility of Stroustrup's approach.

### 7.4 Additional Compile-time Operators

Stroustrup proposed several compile-time operators, such as the compile-time typeof operator.[1] In addition to what Stroustrup proposed we found the primitive operator to be handy. This operator determines whether a type is a primitive type or a class type.

## 8 Conclusion

We found the combination of a control file and macro replacement as a powerful technique for generating type-safe C++ code from typeless templates. We have found many features that make this technique flexible. They are conditionals, Boolean and literal meta-variables, indirection normalization, and controllable file I/O and subprocesses.

In comparison with other objected oriented languages, C++ is a very efficient language, thus making it practical for "real world" programming. The lack of good support for container classes has been a cause for concern, however. Hopefully, with techniques like our parameterization language, C++ can be competitive in that arena also.

## 9 Acknowledgments

David Fletcher, Jon Udell, and Carley Williams provided many helpful suggestions for this project. They also wrote templates that tested out many of the concepts contained in our parameterization tool.

## References

- [1] Bjarne Stroustrup. Parameterized Types for C++. In *Proceedings of the USENIX C++ Workshop*, pages 1-18, Denver, CO, October 1988.
- [2] Stanley B. Lippman. *C++ Primer*. Addison Wesley, 1989.
- [3] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.
- [4] Douglas Lea. libg++, The GNU C++ Library. In *Proceedings of the USENIX C++ Workshop*, pages 243-256, Denver, CO, October 1988.
- [5] Guy Steele. *Common Lisp: The Language*. Digital Press, 1984.



## Exception Handling for C++ (revised)

Andrew Koenig  
Bjarne Stroustrup

AT&T Bell Laboratories  
Murray Hill, New Jersey 07974  
ark@europa.att.com  
bs@research.att.com

### ABSTRACT

This paper outlines a design for an exception handling mechanism for C++. It presents the reasoning behind the major design decisions and considers their implications for implementation alternatives. The mechanism is flexible, comparatively safe and easy to use, works in a mixed language execution environment, and can be implemented to run efficiently. Two implementation strategies are described in some detail.

### 1 Introduction

The author of a library can detect errors, but does not in general have any idea what to do about them. The user of a library may know how to cope with such errors, but cannot detect them – or else they would have been handled in the user's code and not left for the library to find.

The primary purpose of the exception handling mechanism described here is to cope with this problem for C++ programs; other uses of what has been called exception handling in the literature are considered secondary. See the references for discussions of exception handling techniques and mechanisms.

The mechanism described is designed to handle only synchronous exceptions, such as array range checks. Asynchronous exceptions, such as keyboard interrupts, are not handled directly by this mechanism.

A guiding principle is that exceptions are rare compared to function calls and that error handling code is rare compared to function definitions. We do not wish to use exception handling as a substitute for more conventional control structures. Further, we do not want to require the vast majority of functions to include code specifically relating to exception handling for a program to benefit from exception handling.

The mechanism described here can be implemented in several different ways. In particular, we outline a portable implementation based on C's `setjmp/longjmp` mechanism and an implementation that has no run-time costs when exceptions are *not* raised.

C++ is designed to coexist with other languages that do not support exception handling. Consequently, we rejected any ideas for exception handling that would have required all functions in a program to be written in C++.

After the presentation of the exception handling scheme we discuss its use compared to 'traditional' error handling techniques. Many details that we felt to be important, yet not essential for the appreciation of the fundamentals of the exception handling scheme, are placed in appendices.

This paper is written to stimulate discussion of exception handling in C++. The mechanisms described here are not part of C++ and might never become part of C++. A previous version of these mechanisms was presented at the *C++ at Work* conference in November 1989<sup>6</sup>. The scheme described here owes much to the discussion generated by that presentation. In particular, the revised scheme provides a greater degree of protection against mistakes by transferring more obligations from the programmer to the language implementation.

## 2 An Example

Suppose that an exception called `xxii` can occur ('be raised') in a function `g()` called by a function `f()`. How can the programmer of `f()` gain control in that case? The wish to 'catch' the exception `xxii` when it occurs and `g()` doesn't handle it can be expressed like this:

```
int f()
{
    try {
        return g();
    }
    catch (xxii) {
        // we get here only if 'xxii' was raised
        error("g() goofed: xxii");
        return 22;
    }
}
```

The *catch-clause*<sup>†</sup> is called an exception handler for the kind of exception named `xxii`. A single block can have handlers for several distinct exceptions; a handler marked by the ellipsis, `...`, picks up every exception not previously mentioned. For example:

```
int f()
{
    try {
        return g();
    }
    catch (xx) {
        // we get here only if 'xx' was raised
        error("g() goofed: xx");
        return 20;
    }
    catch (xxii) {
        // we get here only if 'xxii' was raised
        error("g() goofed: xxii");
        return 22;
    }
    catch (...) {
        // we get here only if an exception
        // that isn't 'xxii' or 'xx' was raised
        error("g() goofed");
        return 0;
    }
}
```

The series of handlers is rather like a switch statement. The handler marked `(...)` is rather like a default. Note, however, that there is no 'fall through' from one *catch-clause* to another as there is from one case to another. An alternative, and more accurate analogy is that the set of handlers looks very much like a set of overloaded functions.

An exception handler is associated with a block and is invoked whenever its exception is raised in that block or in any function called from it. For example, say in the example above that `xxii` wasn't actually raised in `g()` but in a function `h()` called by `g()`:

```
int g() { return h(); }

int h()
{
    catch(xxii());           // raise exception 'xxii'
}
```

<sup>†</sup> The grammar for the exception handling mechanism can be found in Appendix B.

The exception `xxii` would still be handled by the handler in `f()`.

We use the reserved word `catch` both in a *catch-clause* and in a *catch-expression* as above. There is no ambiguity in doing so: a *catch-clause* can be identified because the keyword `catch` is always followed by something in parentheses and then a left brace. Again, there is a strong resemblance to functions: the *catch-expression* looks like a function call and invokes a handler that looks like a function definition. As we will see below, this resemblance is more than superficial.

Note that `g()` might be written in C or in some other language that does not know about C++ exceptions. This implies that a fully general implementation of the C++ exception mechanism cannot rely on decorations of the stack frame, passing of hidden arguments to *all* functions, or other techniques that require cooperation by the compiler for every function called.

Once an exception is handled (by some handler), it is no longer considered raised and other handlers that might exist for that exception will not be invoked. In other words, only the active handler most recently passed by the thread of control will be invoked. For example, here `xxii` will still be caught by the handler in `f()`:

```
int e()
{
    try {
        return f();
    }
    catch (xxii) {
        // ...
    }
}
```

Another way to look at it is that if a statement or function handles a particular exception, then the fact that the exception has been raised and handled is invisible in the surrounding context — unless, of course, the exception handler is somehow programmed to notify something in the surrounding context that the exception occurred.

### 3 Naming of Exceptions

What is an exception? In other words, how does one declare an exception? What do exception names, such as `xxii` above, really identify?

Perhaps the most important characteristic of exceptions is that the code that raises them and the code that catches them is usually written independently. So, whatever an exception is, it must be something that allows separately written programs to communicate.

Another criterion is that it should be possible to define ‘groups’ of exceptions and provide handlers for such groups. For example, it should be possible to specify a handler to cope with all exceptions coming from a major sub-system such as the file system, the stream I/O system, the network software, etc. In the absence of such a mechanism, systems relying on large numbers of exceptions (say, hundreds of exceptions) become unmanageable.

Finally, we want to be able to pass arbitrary, user-defined information from the point where an exception is raised to the point where it is caught.

Two suggestions have been made for C++:

[1] that an exception be a class, and

[2] that an exception be an object.

We propose a combination of both.

### Exceptions as Classes

Suppose an exception is a class. That would allow us to use inheritance to group exceptions. For instance, one can imagine an `Overflow` exception that is a kind of `Matherr` exception that in turn is a kind of exception. Inheritance is a natural way to express such a relationship and this seems to support the notion that exceptions should be classes.

Closer inspection, however, raises a question. Raising an exception passes information from where it is raised to where it is caught. That information must be stored somewhere. This implies that we need a run-time representation of an exception, that is, an object representing the exception. In C++, a class is

a type – *not* an object. Classes do not carry data; objects do. Also, if an exception is a class, it seems natural to declare objects of that class. If the classes themselves are used to express raising and catching of exceptions, what good are the objects?

### Exceptions as Objects

If exceptions are objects, what kind of objects? The simplest possibility is to make them integers whose values identify the kind of exception. The trouble with that is that it is difficult for people working apart to choose such integer values without clashing. I may choose the unlikely number 299793 to represent my exception, but so may you and when some unlucky third person tries to use our programs together trouble starts. It would seem more sensible to use the *identity* (that is, the address) of the exception object, rather than its *value*, to denote the particular exception being raised.

But this raises problems as well. If we use the identity of an object to 'name' an exception, what object do we say we want to catch? The object presumably identifies the kind of exception that interests us, but as that exception hasn't been raised yet, how can the object exist? Alternatively, if what we catch is a pre-existing object, what do we raise? If we raise an external object, how do we do so without guaranteeing trouble in an environment where multiple processes share a single address space? See reference 6 for a C++ exception handling scheme based on object identities.

### Exceptions as Classes and Objects

It appears, therefore that the most natural way to describe exceptions is to *raise* objects and *catch* classes. A copy of the object raised is passed to the handler.

Given that, it appears to be possible to use any class in a *catch-clause* and any object as the argument in a *catch-expression*. For example:

```
// add with overflow checking

struct Int_overflow { // a class for overflow exception handling
    const char* op;
    int operand1, operand2;
    Int_overflow(const char* p, int q, int r):
        op(p), operand1(q), operand2(r) { }
};

int add(int x, int y)
{
    if (x > 0 && y > 0 && x > MAXINT - y
        || x < 0 && y < 0 && x < MININT + y)
        catch(Int_overflow("+", x, y));

    // If we get here, either overflow has
    // been checked and will not occur, or
    // overflow is impossible because
    // x and y have opposite sign

    return x + y;
}
```

If the arguments to this function are out of range, the function creates an `Int_overflow` object to describe what went wrong and makes (a copy of) that object available to a corresponding *catch-clause*. The exception thus raised can be caught like this:



```

main()
{
    int a, b, c;

    cin >> a >> b;
    try {
        c = add(a,b);
    }
    catch(Int_overflow e) {
        cout << "overflow in operation " << e.op << ", operands"
             << e.operand1 << " and " << e.operand2 << "\n";
    }

    // and so on
}

```

When the *catch-clause* is entered, *e* will be a copy of the *Int\_overflow* object that was created inside the *add* function to describe what went wrong.

The parallel to function call is now almost exact: The *catch-expression* passes an *Int\_overflow* object to the *catch-clause* declared to accept objects of class *Int\_overflow*. The usual initialization semantics is used to pass this argument. The type of the object raised is used to select the handler in approximately the way the arguments of a function call is used to select an overloaded function.

Basically, C++ supports the definition of classes and the use of objects of classes. Using a class to describe the kind of exception expected simplifies type checking between a *catch-expression* and a *catch-clause*. Using an object to describe the exception itself makes it easy to pass information safely and efficiently between those two points. Relying on such fundamental and well-supported language concepts rather than inventing some special "exception type" ensures that the full power of C++ is available for the definition and use of exceptions.

#### 4 Grouping of Exceptions

Exceptions often fall naturally into families. For example, one could imagine a *Matherr* exception that includes *Overflow*, *Underflow*, and other possible exceptions.

One way of doing this might be simply to define *Matherr* as a class whose possible values include *Overflow* and the others:

```
enum Matherr { Overflow, Underflow, Zerodivide /* ... */ };
```

It is then possible to say

```

try {
    f();
}
catch (Matherr m) {
    switch (m) {
        case Overflow:
            // ...
        case Underflow:
            // ...
        // ...
    }
    // ...
}

```

C++ uses inheritance and virtual functions to avoid this kind of switch on a type field in other contexts. It is possible to use inheritance in a similar way to describe collections of exceptions too. For example:

```

class Matherr { };
class Overflow: public Matherr { };
class Underflow: public Matherr { };
class Zerodivide: public Matherr { };
// ...

```

Such classification is particularly convenient because of the desire to handle any of a collection of exceptions similarly: it is easy to imagine many reasons why one might want to recover from any kind of Matherr without caring precisely which kind it was. Using inheritance in this way makes it possible to say

```

try {
    f();
}
catch (Overflow) {
    // ...
}
catch (Matherr) {
    // ...
}

```

Here an Overflow is handled specifically, and all other Matherr exceptions will be handled by the general case. Of course, a program that says catch (Matherr) will not know what kind of Matherr it has caught; that is the price of trying to cater to a potentially unknown set of exceptions.

There are cases where an exception can reasonably be considered to belong to two or more groups. For example:

```

class network_file_err : public network_err , public file_system_err { };

```

Obviously, this too is easily handled without language extensions.

### Naming the Current Exception

We name the current exception analogously to naming the argument of a function:

```

try { /* ... */ } catch (zaq e) { /* ... */ }

```

This is a declaration of e with its scope identical to the body of the exception handler. The idea is that e is created using the copy constructor of class zaq immediately before entering the handler. The original exception object is destroyed as a consequence of unwinding the stack to enter the handler.

Thus, to re-raise the particular exception, one writes

```

try { /* ... */ } catch (zaq e) { catch(e); }

```

If the object passed is not just a zaq but an object of a class derived from zaq, only the zaq part is actually passed. This is the usual rule for initialization and argument passing.

Where there is no need to identify the particular exception, there is no need to name the exception object:

```

try { /* ... */ } catch (zaq) { /* ... */ }

```

This again is similar to the technique of not naming unused function arguments in a function definition.

As with functions, the argument declaration (...) accepts any call, so that

```

catch (...) { /* any exception will get here */ }

```

will catch every exception. There is no way of finding out what exception caused entry into such a handler. Were such a way provided, type safety would be compromised and the implementation of exception handling complicated. The scheme that defines the information transfer from the catch expression at the raise-point to the handler as argument passing is simple, easy to implement efficiently, and safe. Complications requiring mechanisms beyond this elegant scheme – such as mechanisms for identifying the exception in a (...) clause or a general re-raise operation – require serious argument.

## Order of Matching

The *catch-clauses* of a *try-statement* are tried in order. For example:

```
try {
    // ...
}
catch (ibuf) {
    // handle input buffer overflow
}
catch (io) {
    // handle any io error
}
catch (stdlib) {
    // handle any library exception
}
catch (...) {
    // handle any other exception
}
```

The type in a catch clause matches if either it directly refers to the exception raised or is of a base class of that exception, or if the exception raised is a pointer and the exception caught is a pointer to a base class of that exception.

Since the compiler knows the class hierarchy it can warn about sillinesses such as a (...) clause that is not the last clause or a clause for a base class preceding a clause for a class derived from it. In both cases the later clause(s) could never be invoked because they were masked.

## No Match

Where no match is found, the search for a handler continues to the calling function. When a handler is found and entered, the exception is no longer considered raised. After dropping off the end of a *catch-clause* the computation continues after the complete *try-statement*. For example,

```
void f()
{
    try {
        // ...
    }
    catch (x) {
        // do nothing
    }
    catch (...) {
        // do nothing
    }

    // whatever happens (short of hardware failure)
    // we'll get here

    h();
}
```

## Exception Types

Any pointer type or type with a public copy constructor can be used to name an exception in a *catch-clause*. When an exception of a derived class is raised directly, the use of the copy constructor implies that the exception raised is 'cut down' to the exception caught. For example:

```

class Matherr { /* .... */ };

class Int_overflow: public Matherr {
public:
    char* op;
    int operand1, operand2;
    // ...
};

void f()
{
    try {
        g();
    }
    catch (Matherr m) {
        // ...
    }
}

```

When the Matherr handler is entered, m is a Matherr object, even if the call to g() raised Int\_overflow.

For some applications, it may be useful to use virtual functions to deal with exceptions whose type is not statically known at the point where the exception is caught. One way to do that is to use pointers as exceptions:

```

void g()
{
    if ( /* ... */ ) catch(new Int_overflow( /* arguments */ ));
}

void f()
{
    try {
        g();
    }
    catch (Matherr* mp) {
        mp->vfun();
        // ...
        delete mp;
    }
}

```

## 5 Handling of Destructors

Consider what to do when an exception is raised and we need to pass 'up' through a function to find a handler. This is commonly called 'stack unwinding.' If a local object is of a class with a destructor, that destructor must be called as part of the stack unwinding. This is done by implicitly providing a handler that calls the destructor when an exception is raised after construction but before destruction of the local object. For example, if classes A and B have destructors,

```

void f()
{
    A a;
    B b;
    g();
}

```

is equivalent to



```

void f()          // pseudo code
{
    try {
        A a;
        try {
            B b;
            g();
        }
        catch (...) {
            // destroy b
            // re-raise the exception that
            // brought us here
        }
    }
    catch (...) {
        // destroy a
        // re-raise the exception that
        // brought us here
    }
    // destroy b
    // destroy a
}

```

This looks horrendously complicated and expensive. Fortunately, a user will never have to write it, and in fact, there is an implementation technique that (in addition to the standard overhead of establishing a handler) involves only minimal overhead. Here is an example of C code that might be generated for the example above:

```

void f()          // generated code
{
    int __counter = 0;
    struct A a;
    struct B b;

    A::A(&a);
    __counter++;          /* one object created */

    B::B(&b);
    __counter++;          /* two objects created */

    g();

    __counter = 3;

__catch:
    switch (__counter) {
        case 3:
        case 2: b.B::~~B();          /* fall through */
        case 1: a.A::~~A();
    }

    if (__counter == 3) return;
    // re-raise
}

```

For this to work we assume two things: a way of getting to the label `__catch` in case an exception is raised in `g()`, and a way of re-raising the exception at the end of `f()` if an exception caused transfer of control to `__catch`.

The problem becomes somewhat more complicated in the face of arrays of objects with destructors, but we do not believe these complications are any more than tedious to deal with.

## Suppressing Destructor Calls

It has been observed that for many objects destruction is irrelevant once an exception is raised and that an exception may be raised precisely under conditions where it would be dangerous to run the destructor for certain objects. One could therefore imagine facilities for specifying that objects of certain classes should not be destroyed during the stack unwinding caused by exception handling. Such facilities should lead to faster and safer exception handling and more compact code.

Nevertheless, lack of experience argues that we should not immediately include specific facilities for that. Where necessary, the user can simulate the effect of such facilities by inserting suitable checks in destructors. Should experience show that such code proliferates, and especially if experience shows such code to be baroque, the question of a facility for selectively suppressing destructor calls should be revisited.

## Raising Exceptions in Destructors

Naturally, a destructor can raise an exception. In fact, since destructors often call other functions that may raise exceptions it cannot be avoided. Raising exceptions during a "normal" invocation of a destructor causes no special problems. Raising an exception during the execution of a destructor that was called as part of the stack unwinding caused by an exception being raised may be another matter. This should be allowed to avoid putting artificial constraints on the writing of destructors.

Only one restriction is needed: If a destructor invoked during stack unwinding raises an exception, directly or indirectly, that propagated uncaught back through that destructor itself, the exception will be turned into a call of `terminate()`. The default meaning of `terminate` is `abort()`; see Appendix H. Without this restriction the implementation of exception handling would be needlessly complicated and it would not at all be clear what exception should be considered raised: the original exception or the one raised by the destructor? We see no general rule for resolving that question.

## 6 Handling of Constructors

An object is not considered constructed until its constructor has completed. Only then will a destructor be invoked for the object. An object composed of sub-objects is constructed to the extent that its sub-objects have been constructed.

Ideally, an operation to create an object should ensure that the object is completely and correctly constructed. Failing that, the operation should behave in such a way that the state of the system after failure is restored to what it was before creation. Ideally, naively written constructors should always achieve one of these alternatives and not leave the object in some 'half-constructed' state.

Consider a class `X` for which a constructor needs to acquire two resources `x` and `y`. This acquisition might fail and raise an exception. Without imposing a burden of complexity on the programmer, we must ensure that the constructor never returns having acquired resource `x` but not resource `y`.

We can use the technique we use to handle local objects. The trick is to use objects of two classes `A` and `B` to represent the acquired resources (naturally a single class would be sufficient if the resources `x` and `y` are of the same kind). The acquisition of a resource is represented by the initialization of the local object that represents the resource:

```
class X {
    A a;
    B b;
    // ...
    X()
        : a(x), // acquire 'x'
          b(y)  // acquire 'y'
    {}
    // ...
};
```

Now, as in the local object case, the implementation can take care of all the bookkeeping. The user doesn't have to keep track at all.

This implies that where this simple model for acquisition of resources (a resource is acquired by

initialization of a local object that represents it) is adhered to, all is well and – importantly – the user need not write explicit exception handling code.

Consider a case that does not adhere to this simple model:

```
class X {
    static int no_of_Xs;
    widget* p;
    // ...
    X() { no_of_Xs++; p = grab(); }
    ~X() { no_of_Xs--; release(p); }
};
```

What happens if `grab()` raises some exception so that the assignment to `p` never happens for some object? The rule guaranteeing that an object is only destroyed automatically provided it has been completely constructed ensure that the destructor isn't called – thus saving us from the `release()` of a random pointer value – but the `no_of_Xs` count will be wrong. It will be the programmer's job to avoid such problems. In this case the fix is easy: simply don't increase `no_of_Xs` until the possibility of an exception has passed:

```
X::X() { p = grab(); no_of_Xs++; }
```

Alternatively, a more general and elaborate technique could be used:

```
X::X()
{
    no_of_Xs++;
    try {
        p = grab();
    }
    catch (grab_failure) {
        no_of_Xs--;
        catch(X_constructor_failure());
    }
}
```

When exceptions are used to signal failure to construct an object, care must be taken that the constructor restores the environment to the state it found it in before raising an exception.

There is no doubt that writing code for constructors and destructors so that exception handling does not cause 'half constructed' objects to be destroyed is an error-prone aspect of C++ exception handling. The 'resource acquisition is initialization' technique can help making the task manageable. It should also be noted that the C++ default memory allocation strategy guarantees that the code for a constructor will never be called if `operator new` fails to provide memory for an object so that a user need not worry that constructor (or destructor) code is executed for a non-existent object.

## 7 Termination vs. Resumption

Should it be possible for an exception handler to decide to resume execution from the point of the exception? Some languages, such as PL/I and Mesa, say yes and reference 7 contains a concise argument for resumption in the context of C++. However, we feel it is a bad idea and the designers of Modula2+, Modula3, and ML agree.

First, if an exception handler can return, that means that a program that raises an exception must assume that it will get control back. Thus, in a context like this:

```
if (something_wrong) catch (zxc());
```

it would not be possible to be assured that `something_wrong` is false in the code following the test because the `zxc` handler might resume from the point of the exception. What such resumption provides is a contorted, non-obvious, and error-prone form of co-routines. With resumption possible, raising an exception ceases to be a reliable way of escaping from a context.

This problem might be alleviated if the exception itself or the raise operation determined whether to resume or not. Leaving the decision to the handler, however, seems to make programming trickier than

necessary because the handler knows nothing about the context of the raise point. However, leaving the resumption/termination decision to the raise point is equivalent to providing a separate mechanism for termination and resumption – and that is exactly what we propose.

Exception handling implies termination; resumption can be achieved through ordinary function calls. For example:

```
void problem_X_handler(arguments)    // pseudo code
{
    // ...
    if (we_can_recover)
        return;
    else
        catch(X());
}
```

Here, an exception that may or may not resume is simulated by a function. We have our doubts about the wisdom of using *any* strategy that relies on conditional resumption, but it is achievable through ordinary language mechanisms provided there is cooperation between the program that raises the exception and the program that handles it. Such cooperation seems to be a prerequisite for resumption. Pointers to such “error handling” function provides yet another degree of freedom in the design of error handling schemes based of the termination-oriented exception handling scheme presented here.

Resumption is of only limited use without some means of correcting the situation that led to the exception. However, correcting error conditions after they occur is generally much harder than detecting them beforehand. Consider, for example, an exception handler trying to correct an error condition by changing some state variable. Code executed in the function call chain that led from the block with the handler to the function that raised the exception might have made decisions that depended on that state variable. This would leave the program in a state that was impossible without exception handling; that is, we would have introduced a brand new kind of bug that is very nasty and hard to find. In general, it is much safer to re-try the operation that failed from the exception handler than to resume the operation at the point where the exception was raised.

If it were possible to resume execution from an exception handler, that would force the unwinding of the stack to be deferred until the exception handler exits. If the handler had access to the local variables of its surrounding scope without unwinding the stack first we would have introduced an equivalent to nested functions. This would complicate either the implementation/semantics or the writing of handlers.

It was also observed that many of the “potential exceptions” that people wanted to handle using a resumption-style exception handling mechanism where asynchronous exceptions, such as a user hitting a break key. However, handling those would imply the complications of the exception handling mechanism as discussed in section 9.

Use of a debugger can be seen as constituting a special form of resumption; see Appendix G.

## 8 Safety and Interfaces

Raising or catching an exception affects the way a function relates to other functions. It is therefore worth considering if such things should affect the type of a function. One could imagine requiring the set of exceptions raised and the set of exceptions raised to be declared as part of the type of a function. For example:

```
void f(int a) catch (x2, x3, x4);
{
    // ...
}
```

This would improve static analysis of a program and reduce the number of programmer errors<sup>3,4</sup>. In particular, one could ensure that no exception could be raised unless there were a matching handler for it.



## Static Checking

Compile and link time enforcement of such rules is ideal in the sense that important classes of nasty run-time errors are completely eliminated (turned into compile time errors). Naturally, such enforcement does not carry any run-time costs – one would expect a program using such techniques to be smaller and run faster than the less safe traditional alternative of relying on run-time checking. Unfortunately, such enforcement also has nasty consequences.

Suppose a function `f()` can raise an exception `e`:

```
void f() catch (e);
{
    // ...
}
```

For static checking to work, it must be an error for a function `g()` to call `f()` unless either `g()` is also declared to raise `e` or `g()` protects itself against `f()` raising `e`:

```
void g() catch (h);
{
    f();           // error: f() might raise e

    try {
        f();       // OK: if f() raises e, g() will raise h
    } catch (e) {
        catch(h);
    }
}
```

To allow such checking *every* function must be decorated with the exceptions it might raise. This would be too much of a nuisance for most people. For example:

```
#include <stream.h>

main()
{
    cout << "Hello world\n"; // error: might raise an exception!
}
```

Preventing such compile-time errors would be a great bother to the programmer and the checking would consequently be subverted. Such subversion could take many forms. For example, calls through C functions could be used to render the static checking meaningless, programmers could declare functions to raise all exceptions by default, and tools could be written to automatically create lists of raised exceptions. The effect in all cases would be to eliminate both the bother and the safety checks that were the purpose of the mechanism.

Consequently, we think the only way to make static checking practical would be to say that undecorated functions can potentially raise any exception at all. Unfortunately, that again puts an intolerable burden on the programmer who wishes to write a function that is restricted to a particular set of exceptions. That programmer would have to guard *every* call to *every* unrestricted function – and since functions are unrestricted by default, that would make restricted functions too painful to write in practice. For instance, *every* function that did not expect to raise I/O exceptions would have to enumerate every possible I/O exception.

## Dynamic checking

A more attractive and practical possibility is to allow the programmer to specify what exceptions a function can potentially raise, but to enforce this restriction at run time only. When a function says something about its exceptions, it is effectively making a guarantee to its caller; if during execution that function does something that tries to abrogate the guarantee, the attempt will be transformed into a call of `terminate()`. The default meaning of `terminate()` is `abort()`; see Appendix H for details.

In effect, writing this:

```
void f() catch (e1, e2);
{
    // stuff
}
```

is equivalent to writing this:

```
void f()
{
    try {
        // stuff
    }
    catch (e1 e) {
        catch(e);
    }
    catch (e2 e) {
        catch(e);
    }
    catch (...) {
        terminate();
    }
}
```

The advantage of the explicit declaration of exceptions in the function type over the equivalent checking in the code is not just that it saves typing. The most important advantage is that the function declaration belongs to an interface that is visible to its callers. Function definitions, on the other hand, are not universally available and even if we do have access to the source code of all our libraries we strongly prefer not to have to look at it very often.

Another advantage is that it may still be practical to detect many uncaught exceptions during compilation. For example, if the first `f()` above called a function that could raise something other than `e1` or `e2`, the compiler could warn about it. Of course the compiler has to be careful about such warnings. For example, if every integer addition can potentially raise `Int_overflow`, it is important to suppress warnings about such common operations. Programmers do not appreciate being showered by spurious warnings.

Detecting likely uncaught exceptions may be a prime job for the mythical `lint++` that people have been looking for an excuse to design: Here is a task that requires massive static analysis of a complete program and does not fit well with the traditional C and C++ separate compilation model.

## Type Checking of Exceptions

Naturally, if a list of exceptions is specified then it is part of a function's type. This implies that the list of exceptions of a declaration must match the list specified in the definition of a function. Like the return type, the exception specification does not take part in function matching for overloaded functions.

A function declared without an exception specification is assumed to raise every exception.

```
int f(); // can raise any exception
```

The opposite case, a function that will raise no exceptions can be declared like this:

```
int g() catch(); // no exception raised
```

The list of exceptions could be part of the function's signature so that type-safe linkage could be achieved. This, however, would require sophisticated linker support so the exception specification is better left unchecked on systems with traditional linker (just as the return type is left unchecked). Simply forcing a re-write and recompilation whenever an exception specification changed would cause the users to suffer the worst implication of strict static checking.

A function must potentially be changed and recompiled if a function it calls (directly or indirectly) made changes to the set of exceptions it caught or raised. This could lead to major delays in the production of software produced (partly) by composition of libraries from different sources since such libraries would *de facto* have to agree on a set of exceptions to be used. For example, if sub-system X handles exceptions from sub-system Y and the supplier of Y introduces a new kind of exception, then

X's code will have to be modified to cope. If I am a user of X and Y, I will not be able to upgrade to a new version of Y until X has been modified. Where many sub-systems are used this can cause cascading delays. Even where the 'multiple supplier problem' does not exist this can lead to cascading modifications of code and to large amounts of re-compilation.

Such problems would cause people to avoid using the exception specification mechanism or else subvert it. To avoid such problems either the exceptions must not be part of a function's signature or else the linker must be smart enough to allow calls to functions with signatures that differ from the expected in their exception specification only.

## 9 Asynchronous Events

Can exceptions be used to handle things like signals? Almost certainly not in most C environments. The trouble is that C uses functions like `malloc` that are not re-entrant. If an interrupt occurs in the middle of `malloc` and causes an exception, there is no way to prevent the exception handler from executing `malloc` again.

A C++ implementation where calling sequences and the entire run-time library are designed around the requirement for reentrancy would make it possible for signals to raise exceptions. Until such implementations are commonplace, if ever, we must recommend that exceptions and signals be kept strictly separate from a language point of view. In many cases, it will be reasonable to have signals and exceptions interact by having signals store away information that is regularly examined (polled) by some function that in turn may raise appropriate exceptions in response to the information stored by the signals.

## 10 Concurrency

One common use of C++ is to emulate concurrency, typically for simulation or similar applications. The 'task library' that AT&T distributes with `cfront` is one example of such concurrency simulation<sup>12</sup>. The question naturally arises: how do exceptions interact with such concurrency?

The most sensible answer seems to be that unwinding the stack must stop at the point where the stack forks. That is, an exception must be caught in the process in which it was raised. If it is not, an exception needs to be raised in the parent process. An exception handled in the function that is used to create new processes can arrange that by providing a suitable exception handler.

The exception handling scheme presented here is easily implemented to pass information from the raise point to handlers in a way that works correctly in a concurrent system; see Appendix A for details. The 'resource acquisition is initialization' technique mentioned in section 6 can be used to manage locks.

## 11 C Compatibility

Since ANSI C does not provide an exception handling mechanism, there is no issue of C compatibility in the traditional sense. However, a convenient interface to the C++ exception handling mechanism from C can be provided. This would allow C programmers to share at least some of the benefits of a C++ exception handling implementation and would improve mixed C/C++ systems. In 'plain' ANSI C this would require tricky programming using the functions implementing the C++ exception handling mechanism directly. This would, however, still be simpler and better than most current C error handling strategies. Alternatively, C could be extended with a `try` statement for convenience, but if one were going that far there would be little reason not to switch completely to C++.

## 12 Standard Exceptions

A set of exceptions will be standard for all implementations. In particular, one would expect to have standard exceptions for arithmetic operations, out-of-range memory access, memory exhaustion, etc. Libraries will also provide exceptions. At this point we do not have a suggested list of standard exceptions such as one would expect to find in a reference manual.

This paper describes a set of language mechanisms for implementing error handling strategies – not a complete strategy.

### 13 So How Can We Use Exceptions?

The purpose of the exception handling mechanism is to provide a means for one part of a program to inform another part of a program that an 'exceptional circumstance' has been detected. The assumption is that the two parts of the program are typically written independently and that the part of the program that handles the exception often can do something sensible about it.

What kind of code could one reasonably expect to find in an exception handler? Here are some examples:

```
int f(int arg)
{
    try {
        g(arg);
    }
    catch (x1) {
        // fix something and retry:
        g(arg);
    }
    catch (x2) {
        // calculate and return a result:
        return 2;
    }
    catch (x3 x) {
        // pass the bug
        catch(x);
    }
    catch (x4) {
        // turn x4 into some other exception
        catch(xxii);
    }
    catch (x5) {
        // fix up and carry on with next statement
    }
    catch (...) {
        // give up:
        terminate();
    }
    // ...
}
```

Does this actually make error handling easier than 'traditional techniques?' It is hard to *know* what works without first trying it, so we can only conjecture. The exception handling scheme presented here is synthesized from schemes found in other languages, but learning from other people's mistakes (and successes) is not easy and what works in one language and for a given set of applications may not work in another language for different range of applications.

Consider what to do when an error is detected deep in a library. Examples could be an array index range error, an attempt to open a non-existent file for reading, falling off the stack of a process class, or trying to allocate a block of memory when there is no more memory to allocate. In a language like C or C++ without exceptions there are only few basic approaches (with apparently infinite variations). The library can

- [1] Terminate the program.
- [2] Return a value representing 'error.'
- [3] Return a legal value and leave the program in an illegal state.
- [4] Call a function supplied to be called in case of 'error.'

What can one do with exceptions that cannot be achieved with these techniques? Or rather – since anything can be fudged given sufficient time and effort – what desirable error handling techniques become easier to write and less error-prone when exceptions are used?

One can consider raising an exception as logically equivalent to case [1]: 'terminate the program' with the proviso that if some caller of the program thinks it knows better it can intercept the 'terminate order' and try to recover. The *default* result of raising an exception is exactly that of termination (or



entering the debugger on systems where that is the default response to encountering a run-time error).

Exception handling is not really meant to deal with 'errors' that can be handled by [4] 'call an error function.' Here, a relation between the caller and the library function is already established to the point where resumption of the program is possible at the point where the error was detected. It follows that exception handling is not even as powerful as the function call approach. However, should the 'error function' find itself unable to do anything to allow resumption then we are back to cases [1], [2], or [3] where exceptions may be of use.

Case [2], 'returning an error value,' such as 0 instead of a valid pointer, NaN (not a number, as in IEEE floating point) from a mathematical function, or an object representing an error state, implies that the caller will test for that value and take appropriate action when it is returned. Experience shows that

- [1] There are often several levels of function calls between the point of error and a caller that knows enough to handle the error, and
- [2] it is typically necessary to test for the error value at most intermediate levels to avoid consequential errors and to avoid the 'error value' simply being ignored and not passed further up the call chain. Even if error values such as NaN can be propagated smoothly, it can be hard to find out what went wrong when a complicated computation produces NaN as its ultimate result.

Where this is the case one of two things happens:

- [1] Sufficient checking is done and the code becomes an unreadable maze of tests for error values, or
- [2] insufficient checking is done and the program is left in an inconsistent state for some other function to detect.

Clearly, there are many cases where the complexity of checking error values is manageable. In those cases, returning error values is the ideal technique; in the rest, exception handling can be used to ensure that [2], 'insufficient checking' will not occur and that the complexity induced by the need for error handling is minimized through the use of a standard syntax for identifying the error handling code and through a control structure that directly supports the notion of error handling. Using exception handling instead of 'random logic' is an improvement similar to using explicit loop-statements rather than `gotos`; in both cases benefits only occur when the 'higher level' construct is used in a sensible and systematic manner. However, in both cases that is easier to achieve than a sensible and systematic use of the 'lower level' construct.

Case [3], 'return a legal value leaving the program in an illegal state,' such as setting the global variable `errno` in a C program to signal that one of the standard math library functions could not compute a valid result relies on two assumptions

- [1] that the legal value returned will allow the program to proceed without subsequent errors or extra coding, and
- [2] someone will eventually test for the illegal state and take appropriate action.

This approach avoids adding complex error handling tests and code to every function, but suffers from at least four problems:

- [1] Programmers often forget to test for the error state.
- [2] Subsequent errors sometimes happen before the execution gets back to the test for the error condition.
- [3] Independent errors modify the error state (e.g. overwrite `errno`) so that the nature of the problem encountered becomes confused before getting back to the error test.
- [4] Separate libraries assign the same values (error states) to designate different errors, thus totally confusing testing and handling of errors.

Another way of describing the exception handling scheme is as a formalization of this way of handling errors. There is a standard way of signaling an error that ensures that two different errors cannot be given the same 'error value,' ensures that if someone forgets to handle an error the program terminates (in whichever way is deemed the appropriate default way), and (since the ordinary execution path is abandoned after an exception is raised) all subsequent errors and confusion will occur in exception handling code – most of which will be written specifically to avoid such messes.

So, does the exception handling mechanism solve our error handling problems? No, it is only a mechanism. Does the exception handling mechanism provide a radically new way of dealing with errors? No, it simply provides a formal and explicit way of applying the standard techniques. The exception handling mechanism

- [1] Makes it easier to adhere to the best practices.
- [2] Gives error handling a more regular style.
- [3] Makes error handling code more readable.
- [4] Makes error handling code more amenable to tools.

The net effect is to make error handling less error-prone in software written by combining relatively independent parts.

One aspect of the exception handling scheme that will appear novel to C programmers is that the default response to an error (especially to an error in a library) is to terminate the program. The traditional response has been to muddle on and hope for the best. Thus exception handling makes programs more 'brittle' in the sense that more care and effort will have to be taken to get a program to run acceptably. This seems far preferable, though, to getting wrong results later in the development process (or after the development process was considered complete and the program handed over to innocent users).

The exception handling mechanism can be seen as a run-time analog to the C++ type checking and ambiguity control mechanisms. It makes the design process more important and the work involved in getting a program to compile harder than for C while providing a much better chance that the resulting program will run as expected, will be able to run as an acceptable part of a larger program, will be comprehensible to other programmers, and amenable to manipulation by tools. Similarly, exception handling provides specific language features to support 'good style' in the same way other C++ features support 'good style' that could only be practiced informally and incompletely in languages such as C.

It should be recognized that error handling will remain a difficult task and that the exception handling mechanism while far more formalized than the techniques it replaces still is relatively unstructured compared with language features involving only local control flow.

For example, exceptions can be used as a way of exiting from loops:

```
void f()
{
    class loop_exit { };

    // ...
    try {
        while (g()) {
            // ...
            if (I_want_to_get_out) catch(loop_exit());
            // ...
        }
    }
    catch (loop_exit) {
        // come here on 'exceptional' exit from loop
    }
    // ...
}
```

We don't recommend this technique because it violates the principle that exceptions should be exceptional: there is nothing exceptional about exiting a loop. The example above simply shows an obscure way of spelling goto.

## 14 Conclusions

The exception handling scheme described here is flexible enough to cope with most synchronous exceptional circumstances. Its semantics are independent of machine details and can be implemented in several ways optimized for different aspects. In particular, portable and run-time efficient implementations are both possible. The exception handling scheme presented here should make error handling easier and less error-prone.

## 15 Acknowledgements

Michael Jones from CMU helped crystallize early thoughts about exception handling in C++ and demonstrate the power of some of the basic exception handling implementation techniques originating in the Clu and Modula2+ projects. Jim Mitchell contributed observations about the problems with the Cedar/Mesa and Modula exception handling mechanisms. Bill Joy contributed observations about interactions between exception handling mechanisms and debuggers. Dan Weinreb provided the 'network file system exception' example.

Jerry Schwarz and Jonathan Shopiro contributed greatly to the discussion and development of these ideas. Doug McIlroy pointed out many deficiencies in earlier versions of this scheme and contributed a healthy amount of experience with and skepticism about exception handling schemes in general. Dave Jordan, Jonathan Shopiro, and Griff Smith contributed useful comments while reviewing earlier drafts of this paper.

Discussions with several writers of optimizing C compilers were essential.

A discussion in the `c.plus.plus/new.syntax` conference of the BIX bulletin board helped us improve the presentation of the exception handling scheme.

The transformation of the exception handling scheme presented at the C++ at Work conference into the scheme presented here owes much to comments by and discussions with Toby Bloom, Dag Brück, Peter Deutsch, Keith Gorlen, Mike Powell, and Mike Tiemann. The influence from ML is obvious.

We are also grateful to United Airlines for the period of enforced airborne idleness between Newark and Albuquerque that gave us the time to hatch some of these notions.

## 16 References

- [1] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson: *Modula-3 Report*. DEC Systems Research Center. August 1988.
- [2] J. Goodenough: *Exception Handling: Issues and a Proposed Notation*. CACM December 1975.
- [3] Steve C. Glassman and Michael J. Jordan: *Safe Use of Exceptions*. Personal communication.
- [4] Steve C. Glassman and Michael J. Jordan: *Preventing Uncaught Exceptions*. Olivetti Software Technology Laboratory. August 1989.
- [5] Griswold, Poage, Polonsky: *The SNOBOL4 Programming Language*. Prentice-Hall 1971
- [6] Andrew Koenig and Bjarne Stroustrup: *Exception Handling for C++*. Proc. C++ at Work Conference, SIGS Publications, November 1989.
- [6] Bertrand Meyer: *Object-oriented Software Construction*. Prentice Hall. 1988.
- [7] Mike Miller: *Exception Handling Without Language Extensions*. Proceedings of the 1988 USENIX C++ Conference.
- [8] B. Milner, M. Tofte, R. Harper: *The Definition of Standard ML*. MIT Press 1990.
- [9] James G. Mitchell, William Maybury, and Richard Sweet: *Mesa Language Manual*. Version 5.0. April 1979. XEROX PARC CSL-79-3.
- [10] Barbara Liskov and Alan Snyder: *Exception Handling in CLU*. IEEE ToSE. November 1979.
- [11] Paul Rovner: *Extending Modula-2 to Build Large, Integrated Systems*. IEEE Software Vol.3 No.6 November 1986. pp 46-57.
- [12] Jonathan Shopiro and Bjarne Stroustrup: *A Set of C++ Classes for Co-Routine Style Programming*. Proc USENIX C++ Workshop. Santa Fe, NM. November 1987.
- [13] Sun Common Lisp Advanced User's Guide. Section 3: *The Error Handling Facility*.
- [14] David Waitzman: *Exception Handling in Avalon/C++*. CMU Dept. of Computer Science. August 6. 1987.
- [15] USA Department of Defense: *Reference Manual for the ADA Programming Language*. ANSI/MID-STD-1815A-1983.
- [16] Shaula Yemini and Daniel M. Berry: *A Modular Verifiable Exception-Handling Mechanism*. ACM ToPLaS. Vol.7, No.2, April 1985, pp 214-243.

## 17 Appendix A: Implementations

We have described the design of a set of exception handling mechanisms. Several different styles are possible. In particular, it is possible to write an implementation that is fully portable in the sense that it produces ANSI C (only) and also to write an implementation that is close to ideal in its use of run-time.

### Run-time representation of Types

All implementations must provide a way of representing the type of the exception raised for matching with the types specified in the handlers. At the point where an exception is raised, the compiler might pass the library functions implementing the search for a handler a pointer to a string that names the exception class and all its base classes, along with a pointer to the copy constructor for each. A variant of the encoding of types used to achieve type-safe linkage on systems with traditional linkers could be used. For example:

```
class X { /* ... */ };

class Y : public X { /* ... */ };

class A { /* ... */ };

class Z : public Y : public A { /* ... */ };
```

could be represented as:

type	run-time representation
X	"_1X"
Y	"_1Y: _1X"
Z	"_1Z: ( _1Y: _1X, _1A) "
Y&	"R_1Y: _1X"
Z*	"P_1Z: ( _1Y: _1X, _1A) "
const char*	"PCc"

The reason that strings are a reasonable representation and that the complete inheritance hierarchy is needed is that separate compilation of the *catch-expression* and the *catch-clause* will be the norm and that the resolution must be done at run time.

Schemes based on other kinds of objects representing the classes are of course also feasible and will be more appropriate than strings in some environments. Whatever representation is chosen should be suitably encapsulated in a class. It will also be necessary to define a set of C functions for creating and using such run-time representations. We present the string encoding scheme simply as an existence proof.

### Outline of a Portable Implementation

Here we describe an implementation that is fully portable in the sense that it makes it possible to generate ANSI C (only) from a C++ program. The importance of such an implementation is that it allows programmers to experiment with C++ exception handling without waiting for implementations to be specifically tuned for their particular system and guarantees portability of programs using exception handling across a greater range of machines than would be economically feasible had each machine required a 'hand crafted' implementation. The price of such an implementation is run-time cost. We have no really good estimates of that cost but it is primarily a function of the cost of `setjmp()` and `longjmp()` on a given system.

To simplify the discussion, we need to define a few terms. A *destructible object* is an object that has a non-empty destructor or contains one or more destructible objects. The *destructor count* of a class is the number of destructors that must be executed before freeing an object of that class, including the destructor for the class itself. The destructor count of an array is the number of elements in the array times the destructor count of an element.



The general strategy is to lay down a destructor for every destructible object, even if the user did not specify one (for example, consider an object without an explicit destructor that contains two objects with destructors). Every destructible object on the stack acquires a *header* containing a pointer to the destructor for that object, a pointer to the next most recently created automatic destructible object, and a *skip count* that indicates how much of that object has been constructed:

```
struct _header {
    void* destructor;
    void* backchain;
    unsigned skipcount;
};
```

Thus, for example, a C++ class that looks like this:

```
class T {
public:
    T();
    ~T();
};
```

will be augmented internally to look like this:

```
class T {
    _header _h;
public:
    T();
    ~T();
}
```

Every `_header` object is part of some other object. All destructible objects on the stack therefore form a chain, with the most recent first. The head of that chain is a (single) global variable created for the purpose, initially null.

Therefore, to construct an automatic object, the implementation must make the object header point to the destructor for that object, set the skip count to indicate that construction has not yet started, and link the header into the global chain. If the variable that heads the chain is called `_DO_head` (DO stands for 'destructible objects'), the following code would be inserted into `T::T()` to maintain the chain:

```
_h.backchain = _DO_head;
_h.skipcount = 1; // destructor count of T
_h.destructor = (void*) &T::~~T();
_DO_head = &h;
```

We will use this chain *only* to destroy objects while unwinding the stack; we do not need it when a destructor is called during normal exit from a block. However, an exception could occur in a destructor during normal block exit, so it is necessary to keep the global chain header up to date while destroying objects. The safest way to do this is probably for the compiler to avoid following the actual chain. Instead, the compiler should point the chain header directly at each object it is about to destroy.

In other words, rather than generating code inside `T::~~T()` to update `_DO_head`, the compiler should reset `_DO_head` explicitly and then call the relevant destructor where possible:

```
_DO_head = previous value;
t->T::~~T();
```

The skip count indicates how many destructors have to be skipped when destroying an object. When we are about to begin constructing an object, we set its skip count to its destructor count. Each constructor decrements the skip count, so that a completely constructed object has a skip count of zero.

The elements of an object are destroyed in reverse order. Each destructor *either* decrements the skip count *or* destroys its associated object if the skip count is already zero.

To do all this, we give every constructor and destructor the address of the skip count as an 'extra' argument. If the object is not on the stack, we pass the address of a dummy word.

For example, code generated for the destructor `T::~~T()` might look like this:

```

T_dtor(/* other arguments */, unsigned* skipcount) /* generated code */
{
    if (*skipcount)
        --*skipcount;
    else {
        /* destroy 'this' */
    }
}

```

If destructors leave a zero skip count untouched, we can safely give a destructor for an off-stack object the address of a static zero word somewhere. Having a destructor do this costs little, as destructors must test for zero anyway.

The test is presumably unnecessary for constructors, and it would be a shame to have to include it anyway just to avoid having a constructor for an off-stack object scribble a global value. However, we can use a different global value for constructors. We don't care what the value is, so all constructors can safely scribble it. By making all skip counts unsigned, we can avoid the possibility of overflow or underflow.

The actual stack unwinding must be done by `longjmp` and takes place in two stages. First we track back through the stack object chain, calling destructors along the way. After that, we execute the C `longjmp` function to pop the stack.

What is the overhead of this scheme? Every constructor gets an extra argument (which takes one instruction to push it) and has to decrement the skip count (one or two instructions, two memory references). Every destructor gets an extra argument (one instruction to push it) and has to decrement and test the skip count (three or four instructions). In addition, each destructible object gets three extra words (which takes three instructions to fill them) and the list head must be updated when the object is created (one instruction). There is no extra overhead for classes, such as `Complex`, that do not have destructors.

Thus the overhead per destructible object is somewhere near 10-12 instructions. This overhead decreases if constructors or destructors are inline: the skip counts can be set directly instead of indirectly and the inline destructors don't even have to test them. It is still necessary to lay down an out-of-line destructor with full generality.

### Outline of an Efficient Implementation

Here we outline an implementation technique (derived from Clu and Modula2+ techniques) that has close to ideal run-time performance. This means that (except possibly for some table building at startup time), the implementation carries absolutely no run time overhead until an exception is raised; every cycle spent is spent at the point of raising an exception. This makes raising an exception relatively expensive, but that we consider a potential advantage because it discourages the use of exception handling where alternative control structures will do.

Since this implementation requires detailed knowledge of the function calling mechanism and of the stack layout on a particular system, it is portable only though a moderate amount of effort.

The compiler must construct, for each block, a map from program counter values to a 'construction state.' This state is the information necessary to determine which objects need to be destroyed if execution is interrupted at the given point.

Such a table can be organized as a list of address triples, the first two elements representing a region in which a particular destruction strategy is appropriate, and the third element representing a location to which to jump if execution is interrupted in that region. The number of entries in such tables could be greatly reduced (typically to one) at a modest execution cost by adding a fourth element to entries and using the `__counter` trick presented in section 5.

This table is searched only if an exception is raised. In that case, the exception dispatcher unwinds the stack as necessary, using the appropriate table for each stack frame to decide how to destroy objects for that stack frame. The dispatcher can regain control after each level of unwinding by 'hijacking' the return address from each frame before jumping to the destruction point.

It is possible to merge all pre-frame exception tables into a single table either at run time or compile time. Where dynamic linking is used such table merging must be done at run time.

As before, the decision about whether to enter an exception handler can be made by code generated for the purpose, the address of which can be placed in the table in place of the destruction point for those blocks with exception handlers.

The only overhead of this scheme if exceptions are not used is that these tables will take up some space during execution. Moreover, the stack frame layout must be uniform enough that the exception dispatcher can locate return addresses and so on.

In addition, we need some way of identifying these tables to the exception dispatcher. A likely way to do that while still living with existing linkers is to give the tables names that can be recognized by a suitable extension to the munch or patch programs.

## Appendix B: Syntax

In the C++ grammar, the exception handling mechanism looks like this:

```

try-statement:
    try { statement-listopt } catch-clause-list

catch-clause-list:
    catch-clause catch-clause-listopt

catch-clause:
    catch exceptionopt { statement-listopt }

exception:
    ( argument-declaration )
    ( ... )

catch-expression:
    catch ( expression )

```

A *try-statement* is a *statement*. A *catch-expression* is an *expression* of irrelevant type, because it never returns.

A *exception-specification* can be used as a suffix in a function declaration:

```

exception-specification:
    catch ( type-listopt )

type-list:
    type
    type-list , type

```

## Appendix C: Local Scope Issues

Allowing the exception handling code in a *catch-clause* to access variables in the *try* block to which it is attached would violate the notion that a block establishes a scope. For example:

```

void f(int arg)
{
    int loc1;

    try { int loc2; /* ... */ g(); } catch (x1) { /* ... */ }
    try { int loc3; /* ... */ g(); } catch (x2) { /* ... */ }
}

```

Here, both exception handlers can access the local variable *loc1* and the argument *arg*, but neither of them can get at *loc2* or *loc3*.

Disallowing access to variables in the block to which the handler is attached is also important to allow good code to be generated. For example:

```

void f(int arg)
{
    int loc1;

    // ...

    try {
        int loc2 = 1;
        g1();
        loc2++;
        g2();
    }
    catch (x1) {
        if (loc2 == 1) // ...
    }
}

```

Were this allowed, the compiler and run-time systems would have to ensure that the value of `loc2` is well defined upon entry to the handler. On many architectures, this implies that its value must be allocated in memory before the calls of `g1()` and `g2()`. However, this involves an increase of the size of the generated code and poor use of registers. One could easily lose a factor of two in run-time performance while at the same time enlarging the code significantly.

Access to local variables in the surrounding scope (including the function arguments) is allowed and is important in the cases where recovery is possible. For example:

```

void f(int arg)
{
    // ...

    try {
        // ...
    }
    catch (x1) {
        // fix things
        return f(arg); // try again
    }
}

```

When re-trying this way, one should beware of the infinite recursion that can occur in case of repeated failure.

Naturally, the value of local variables accessible from a handler, such as `arg`, must be correct upon entry of the handler. This implies compilers must take a certain amount of care about the use of registers for such variables.

#### Appendix D: Syntax Details

It might be possible to simplify the

```
try { ... } catch (abc) { ... }
```

syntax by removing the apparently redundant `try` keyword, removing the redundant parentheses, and by allowing a handler to be attached to any statement and not just to a block. For example, one might write:

```

void f()
{
    g(); catch (x1) { /* ... */ }
}

```

instead of



```

void f()
{
    try { g(); } catch (x1) { /* ... */ }
}

```

The added notational convenience seems insignificant and may not even be convenient. People seem to prefer syntactic constructs that start with a prefix that alerts them to what is going on, and it may be easier to generate good code when the `try` keyword is required. For example,

```

void f(int arg)
{
    int i;
    // ...
    try { /* ... */ } catch (x) { /* ... */ }
    // ...
}

```

Here, the `try` prefix enables a single-pass compiler to ensure, at the beginning of the block with a handler, that local variables in the surrounding block are not stored in registers. Without the prefix, register allocation would have to be postponed until the end of the function. Naturally, optimal register allocation based on flow analysis does not require `try` as a hint, but it is often important for a language design to enable simple code generation strategies to be applied without incurring devastating run-time penalties.

Allowing exception handlers to be attached to blocks only and not to simple statements simplifies syntax analysis (both for humans and computers) where several exceptions are caught and where nested exception handlers are considered (see Appendix E). For example, assuming that we allowed *catch-clauses* to be attached to any statement we could write:

```
try try f(); catch (x) { ... } catch (y) { ... } catch (z) { ... }
```

This could be interpreted in at least three ways:

```

try { try f(); catch (x) { ... } } catch (y) { ... } catch (z) { ... }

try { try f(); catch (x) { ... } catch (y) { ... } } catch (z) { ... }

try { try f(); catch (x) { ... } catch (y) { ... } catch (z) { ... } }

```

There seems to be no reason to allow these ambiguities even if there is a trivial and systematic way for a parser to choose one interpretation over another. Consequently, a `{` is required after a `try` and a matching `}` before the first of the associated sequence of `catch` clauses.

The parentheses around the exception expression are required in parallel to the requirement of parentheses around the list of formal parameters in a function definition.

The parentheses around the expression in a *catch-expression* are redundant. If `return` were chosen as a syntactic model for the *catch-expression* rather than the function call one would leave them out. Possibly both forms should be allowed:

```

catch new Int_overflow( /* arguments */ );

catch(new Int_overflow( /* arguments */ ));

```

The keyword `catch` is used in three separate ways:

- [1] To declare an exception handler: `catch (ex e) { /* ... */ }`
- [2] To raise an exception: `catch(ex(2));`
- [3] To specify the exceptions raised by a function: `void f(int) catch(a,b,c);`

Would confusion be avoided by introducing three different keywords?:

- [1] To declare an exception handler: `handle (ex e) { /* ... */ }`
- [2] To raise an exception: `raise(ex(2));`
- [3] To specify the exceptions raised by a function: `void f(int) raises(a,b,c);`

We are not sure. The use of the single keyword `catch` reflects the usual state of affairs for functions where a single name is used to declare, define, and call a function. It reflects the common English use

of the word `catch` where one yells *catch!* when throwing an object for someone to catch. Finally it minimizes the number of new keywords introduced.

## Appendix E: Nested Exceptions

What happens if an exception is raised while an exception handler is executing? This question is more subtle than it appears at first glance. Consider first the obvious case:

```
try { f(); } catch (e1 e) { catch(e); }
```

This is the simplest case of raising an exception inside an exception handler and appears to be one of the more common ones: a handler that does something and then passes the same exception on to the surrounding context. This example argues strongly that re-raising `e1` should not merely cause an infinite loop, but rather that the exception should be passed up to the next level.

What about raising a different exception?

```
try { f(); } catch (e2) { catch(e3); }
```

Here, if exception `e2` occurs in `f()`, it seems to make sense to pass `e3` on to the surrounding context, just as if `f()` had raised `e3` directly. Now, let's wrap a block around this statement:

```
try {
    try {
        f();
    }
    catch (e2) {
        catch(e3);
    }
    catch (e3) {
        // inner
    }
}
catch (e3) {
    // outer
}
```

If `f()` raises `e2`, this will result in the handler for `e3` marked 'outer' being entered.

From the language's perspective an exception is considered handled at the point where control is passed to a user-provided handler. From that point on all exceptions are handled by (lexically or dynamically) enclosing handlers.

The handling of destructors naturally gives rise to a form of nested handlers – and these *must* be handled. Nested handlers may naturally occur in machine generated code and could conceivably be used to improve locality of error handling.

## Appendix F: Arguments to Raise Operations

We expect that it will be commonplace to pass information from the point where an exception is raised to the point where it is handled. For example, in the case of a vector range exception one might like to pass back the address of the vector and the offending index. Some exception handling mechanisms provide arguments for the raise operation to allow that. However, since (in the scheme presented here) exceptions are simply objects of user-defined types, a user can simply define a class capable of holding the desired information. For example:

```

class Vector {
    int* p;
    int sz;

public:
    class Range {
        Vector* id;
        int index;
    public:
        Range(Vector* p, int i): id(p), index(i) { }
    };

    int& operator[] (int i)
    {
        if (0<=i && i<sz) return p[i];
        catch(Range(this, i));
    }
};

void f(Vector& v)
{
    try {
        do_something(v);
    }
    catch (Vector::Range r) {
        // r.id points to the vector
        // r.index is the index
        // ...
    }
}

```

This solution works nicely even under a system with concurrency. Raising an exception involves creating a new object on the stack; catching it involves unwinding the stack to the catch point after copying the exception object into the handler. Neither of these operations is a problem for a system that can handle concurrency at all.

## Appendix G: Debuggers

There is one special case where a form of resumption is appropriate and must be supported. On a system with a debugger that allows modification of the state of a running program, one would expect an uncaught exception to cause entry into the debugger. The programmer might then use the debugger change the state of the program or even the program itself and request resumption at the point where the exception was raised. Such a debugger would not be particularly useful if it were entered only after the stack had been completely unwound.

This case requires special attention, but is fortunately quite simple to handle. The internal library code that raises exceptions will look something like this:

```

if (debugger_active() && only_default_handler_exists()) {
    call_debugger();
    return;
}
else {
    // unravel stack and invoke handler
}

```

The test `only_default_handler_exists()` is potentially quite expensive. We consider that perfectly acceptable since we would not like to see exceptions be used for high frequency problems. In time-critical applications we do not expect to have a sophisticated debugger active. Consequently, `only_default_handler_exists()` will not be called and cannot affect performance in those cases. The cost of calling `only_default_handler_exists()` is likely to depend strongly on the

locality of a handler: in the schemes we can think of, discovering that there is no handler will be far more expensive than figuring out that a handler is available in the current function or its caller.

#### Appendix H: `terminate()`

Occasionally, exception handling must be abandoned for less subtle error handling techniques. Examples of this are when the exception handling mechanism cannot find a handler for a raised exception, when the exception handling mechanism finds the stack corrupted, and when a destructor called during stack unwinding cause by an exception tries to exit using an exception. In such cases

```
void terminate();
```

is called.

This function in turn executes the last function given as an argument to the function `set_terminate()`:

```
typedef void(*PFV)();  
PFV set_terminate(PFV);
```

The previous function given to `set_terminate()` will be the return value; this enables users to implement a stack strategy for using `terminate()`:

```
class STC {      // store and reset class  
    PFV old;  
public:  
    STC(PFV f) { old = set_terminate(f); }  
    ~STC() { set_terminate(old); }  
};  
  
void my_function()  
{  
    STC xx(&my_terminate_handler);  
  
    // my_terminate_handler will be called in case of disasters here  
  
    // the destructor for xx will reset the terminate handler  
}
```

This is an example of using a local object to represent the acquisition and relinquishing of a resource as mentioned in section 6. Without it, it guaranteeing the reset of the function to be called by `terminate()` to its old value would have been tricky. With it, `terminate()` will be reset even when `my_function()` is terminated by an exception.

We avoided making `STC` a local class because with a suitable name it would be a good candidate for a standard library.

By default `terminate()` will call `abort()`. That default is expected to be the correct choice for most users.



# *Experiences in Writing a Distributed Particle Simulation Code in C++*

David W. Forslund, Charles Wingate, Peter Ford, J. Stephen Junkins, Jeffrey Jackson, Stephen C. Pope

Los Alamos National Laboratory

Los Alamos, NM 87545

dwf@lanl.gov

## **Abstract**

Although C++ has been successfully used in a variety of computer science applications, it has not yet become of widespread use in scientific applications. We believe that the object-oriented properties of C++ lend themselves well to scientific computations by making maintenance of the code easier, by making the code easier to understand, and by providing a better paradigm for distributed memory parallel codes. We describe here our experiences using C++ to write a particle plasma simulation code using object-oriented techniques for use in a distributed computing environment. We initially designed and implemented the code for serial computation and are in the process of making it work on top of the distributed programming toolkit ISIS. In this connection we describe some of the difficulties presented by using C++ for doing parallel and scientific computation. In the spirit of most C++ papers, we advocate some changes in the language, although we remain devotees of C++ despite the shortcomings cited.

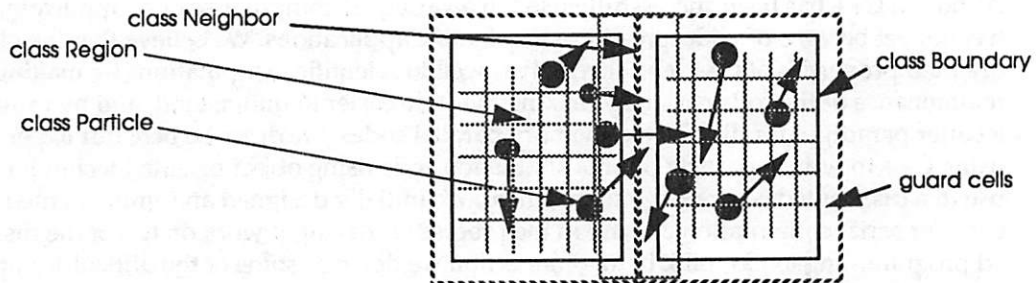
## **Introduction**

Plasma particle simulation involves the modeling of the behavior of an ionized gas in its self-consistent and externally imposed electromagnetic fields by advancing a large collection of particles with discretized Newton's laws on a grid and the solution of discrete electromagnetic field equations using particle quantities interpolated to the grid. WAVE [Forslund 1985] is a moderately large (20,000+ lines) plasma simulation code written in Fortran which is in wide spread use at plasma research sites around the world. The code can consume enormous amounts of computer time in the process of modeling problems ranging from the interaction of intense laser light with ionized gases to the behavior of the interstellar medium. Although the model is quite simple, it requires vast computer resources and is reasonably well suited to running on a massively parallel MIMD computer. There are many physics problems being studied at Los Alamos National Laboratory which will eventually require harnessing several Cray class computers together over high speed channels. Although the channels will be very high speed they will be high latency relative to the clock cycle of the machines. This is similar to a network of RISC based machines running on a ethernet. The goal of this project is to write WAVE++ in an object-oriented manner to simplify and accelerate its use on large parallel computers and simultaneously match the numerical model more closely to the physical world. This also provides a good platform for evaluating the usefulness of C++ for scientific computing and observing some ways in which the language might be improved.

## Design of WAVE++

C++ enables us to design the code in a more modular fashion than is possible to do with Fortran and allows for much more flexibility in the type of data structures used. The most compelling reason to use C++, however, is the natural decomposition of the problem for parallelization. The data and the methods are kept together providing all the information for advancing the particles and fields in the local grids. The grid domain in WAVE++ is currently decomposed with a quadtree in 2 dimensions and an octtree in 3 dimensions. Each leaf (a minimum size grid region) has all the methods and data (particles and fields) needed to solve the problem locally. Communication between the regions is constrained to a single layer of cells surrounding each region. The electromagnetic field equations involve a coupled set of second order partial differential equations. Although in the Fortran version the field equations are solved by block diagonal linear algebra methods, in order to minimize communication in WAVE++, we solve the field equations by means of a local iterative solve with only a small amount of communication across the boundary layers (Fig 1.).

FIGURE 1. Grid and Particles in the Wave++ Model.



Although inheritance is not heavily used in this application, it is quite useful for several of the container classes used to manipulate the data. The basic classes involved in WAVE++ are illustrated in Table I. The two most important are Particle, which contains the methods for advancing the particles in response to the fields, and Region, which is a restricted domain of the grid which contains the fields and particles.

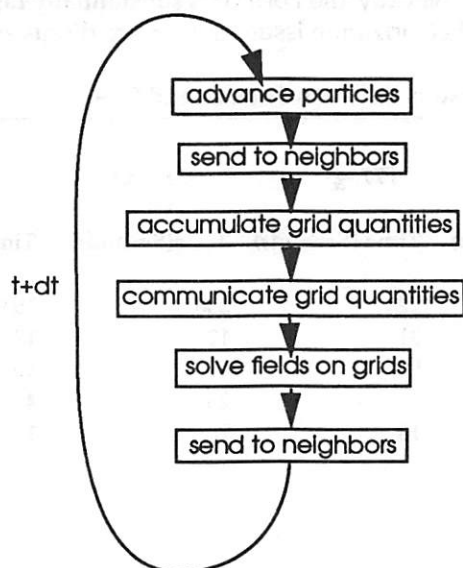
TABLE 1. Class Structure for Wave++.

Class	Functionality
Array	Holds grid variables, provides guard cell references
Boundary	Encapsulates geometry of a region
List	Linked list class
Neighbor	Region guard cells and communication
Particle	Physics of particles, how to accelerate, etc.
Region	Basic physics (contains multiple arrays of fields and currents, knows neighbors, pushes particles, advances fields, low level diagnostics)
Species	Defines properties of groups of particles
Wave	Master for all regions, initializes and collects data

A Region is intended to reside entirely on a single processor of some parallel computer. As the particles move around on the grid, they pass from one region to another, and the fields communi-

cate across the boundaries of the various regions. The class Neighbor, which represents the overlap between adjoining regions, is designed to provide the basic communication buffering between regions. This is done in a memory efficient manner and in such a way as to minimize the number of communication events between regions. Data is accumulated by these buffering classes and is transmitted in one command once the data has been fully assembled. By defining reference Arrays in Neighbor, the update of data in the internal Arrays of Region automatically updates the Array buffers in Neighbor. The particles crossing the region boundaries are also buffered for a single transmission to the appropriate neighboring region. This buffering facilitates the functioning of the code in a distributed, high-bandwidth, high latency environment. We note that the design decisions which were made for the sake of parallelism actually clarified the overall design and increased the speed of the serial implementation. To better illustrate how the physics is related to the communication process, we show the cyclic process of advancing the fields and particles in Fig. 2.

**FIGURE 2. Flow Diagram of Simulation Time Step.**



With the proven capability of C++ in handling graphics, we have designed into the code a modern graphical user interface to assist in the setup of problems and in the interpretation of data generated. This is complicated by the desire to run the code in a distributed parallel environment. We have chosen to use the XView toolkit because of our familiarity with Sunview.

The design of the particle and region classes allows for relatively small effort in changing the code from 2D to 3D (a change in the tree structure and in the boundary class) and in enabling the grid to adaptively reshape itself during the computation to track the physics of the problem. This will allow the code to use a variety of popular simulation methods.

## Serial Wave++ Implementation

WAVE++ was first built in a serial manner on a Sun workstation, in which the leaf regions are explicitly advanced using the generating quadtree as the computational harness. This allowed for a simple debugging environment and basic optimization issues to be addressed and solved. As with most first time C++ programmers, it took a few stabs into dark alleys to discover that the key to successful programming in C++ requires an object oriented design in the first place.

We use gprof++ as provided by Sun Microsystems's C++ 2.0 environment as the performance profiler. In Table 2 we show a comparison of Fortran (F77) and C++ performance times for several categories of a computation with 15,000 particles running 133 time steps. In the Fortran version "Communication" refers to the functions required to bring blocks of particles into the particle mover. In C++ it is the time spent in the routines which send the data between regions and in the link list functions. Note that the Particles and Fields combine to be more than 70% of the computation time in both cases but that the Fortran is about 1.7 times faster than the C++ code. Some of this is due to the much finer granularity of the C++ classes than the Fortran. In the Fortran, one subroutine call pushes 256 particles. In C++ there are multiple methods invoked for each particle. The difference is even greater in the optimized code (factor of 2.2) because of the greater opportunity for optimization in the larger blocks of code and the small function call overhead. Use of inlining would probably help. We are continuing to look at the class hierarchy in an effort to allow greater optimization in C++ without the sacrifice of the object-oriented character. This issue of being able to optimize the C++ code may continue to be a problem unless the compilers become much smarter. In a vector machine such as the Cray, the Fortran is substantially faster because most of the inner loops are fully vectorized. Vectorization issues in C++ are discussed later in this paper.

**TABLE 2. Performance Comparisons between Fortran and C++**

	C++ -g	F77 -g	C++ -O	F77 -O
Function	Time (Seconds)	Time (Seconds)	Time(Seconds)	Time(seconds)
Particles	374	223	338	153
Fields	18	31	12	17
Communication	50	56	40	15
Diagnostics	29	5	25	4
Other	79	10	70	1

## Parallelization Efforts

The granularity of the parallelism can be readily controlled by design, and in general, a number of parallel threads run on a single processor simultaneously. This is primarily done in order to simplify the problem of load balancing between processors by combining together on one node both lightly and heavily loaded threads. At this time we have not attempted to perform dynamic load balancing, although the structure of the code will permit dynamic reconfiguration of the grid based on particle count, which would be a step in this direction.

One of the primary reasons for redesigning and recoding Wave using C++ was to use an object oriented design (OOD) methodology. By using OOD we felt it would be significantly easier to introduce parallelism into the code. The Presto [Bershad et al. 1988] model of computation allows the programmer to explicitly introduce parallelism by creating separate threads of control and then direct each thread to execute an object's method. Using this model allowed us to develop the code in a serial fashion and then add parallelism where we felt it was needed, without disturbing the original code and class hierarchy.



The Presto model of programming in C++ works with a Thread class defined as:

```
class Thread {
private: // internal representation of thread
public:
    Thread();
    call(POBJany, PFany, ...); // execute method on object
}
```

The basic idea is that the user can have the thread execute any method of any object:

```
class Region {
public:
    Region ();
    int update(int, float); // ...
}

Thread *t;
t = new Thread; // create a thread, on standby
Region o;
t.call(o, o.update, 2, 5.4);
```

This example illustrates there are many problems with the C++ implementation of the Presto model, most of which are due to abuse of C++. The current implementation of Presto under C++ can only use methods which are not overloaded, return values are ignored and the argument list is not coerced to the types of the formal arguments of method `Obj::update`. C++ does not have a way of specifying the linkage between the object/method/argument list which would allow proper type checking and coercion. As it stands, `Thread::call` is type unsafe. Note that `Thread::call` is in spirit a polymorphic function, which takes an object of any type `TY`, and a method of `TY` which has a signature to which the argument list can be coerced. Currently C++ does not have semantics (or for that matter syntax) which allow the programmer to express anything about method invocation (or function calling) beyond actually doing the invocation. A bit more formally [Cardelli 1985], let `TY` be a type, `TY:M` a method `M` of type `TY`, and `ARGLIST` a cartesian product over the types in the language. The signature of `Thread::call` is `TY*TY:M*ARGLIST` where `ARGLIST` conforms to the signature of `TY:M`. The result type of `Thread::call` is the result value of `TY:M`. In operational terms, do the standard C++ type check, select the method, generate the correct argument list (coercion), BUT do not invoke the method. In current C++ and the proposed Template extensions by Stroustrup [Stroustrup 1989], the type checking is done only if method invocation syntax is used. What if the underlying system is written in C as most thread systems are (SunOS, Taos, uSystem, etc.)? We might need an interface into something like:

```
thread_invoke(o.update, nargs, <ptr to obj>, <arglist>);
```

At no point in the template is C++ invoking the method and thus a type check is not being done. While it is common for programming languages to ignore issues like these, C++ should take the lead in the Unix community for better software engineering practices. Perhaps it may require the introduction of meta-notation similar to the old lint method of annotating code fragments (an undesirable result)

C++'s model of computation assumes a single thread of execution, compilation rather than interpretation and a shared address space. Many of the problems we have encountered can be traced to these assumptions. An issue which comes up in all distributed systems is how to actually do the remote invocation and argument binding for functions, which has no corresponding analog in the C++ model of computing. Having a first class abstraction for invocation and argument lists built into the language would give the programmer a type safe interface to these objects, and facilitate

the process of delegating the invocation to remote agents and marshalling arguments "across the wire". The ability to get a handle on a thread invocation would allow for the implementation of several styles of concurrent programming. Let us assume that C++ is extended with Stroustrup's proposal for parametric types [Stroustrup 1989], and we have some way in C++ for `t.call` to return a handle to an invocation:

```
// assume INVOKE_HANDLE contains a handle on the thread id
template <TY>
class future {

    INVOKE_HANDLE<int> fh;

public:
    operator TY() {
        // if function is done return value else
        // do a thread join(IH.thread_id)
    };
    operator=(INVOKE_HANDLE<int>);
};

class Thread {
    INVOKE_HANDLE<typeof obj::*> call(obj, obj::*, ARGLIST)
}
// * is universally quantified

{
    future<int> futi;
    futi = t.call(o, o.m, args); // o.m returns an int

    // many clock ticks later ...
    int a = futi;                // future<int>::operator TY()
                                // coercion will wait if necessary
}
```

There have been many extensions made to C++ to support distributed computation. Some have proposed new language constructs such as Michael Tiemann's wrappers [Tiemann 1988] which are specifically intended to aid in building distributed programs. We argue that the proliferation of different concurrent computational models will severely stress any language that can not be easily extended, and that the semantics of method invocation should not be fixed in stone as in current implementations of C++ and other sequential programming languages. Programs should be able to get handles on the fundamental building blocks of invocations so that they can manipulate invocations with semantics differing from the current semantics inherited from simple procedure calls. Concurrent object oriented programming supporting classical rpc, futures, asynchronous send/receive, and passive vs. active objects should be expressible in one programming language. Static typing as in C++ should be preserved so programmers understand what they are writing and compilers have sufficient information to do a good job of optimization.

## Distributed Computing with ISIS

In order to run Wave++ in a distributed environment, we use ISIS, a distributed programming toolkit designed and implemented at Cornell [Birman et al. 1987; Birman et al. 1989]. This choice was made to limit the amount of code we need to write, and to take advantage of features such as a simple communication interface which can support broadcast and replicated data, a lightweight task system running within each process, support for replicated service which supports fault tolerance, the ability to perform high speed state transfer upon node failure and recovery, and a uniform name space. ISIS is written in C and has interfaces for C, Fortran, and Lisp. We are currently

in the process of porting Wave++ to ISIS and developing a C++ interface into the fundamental structure of the ISIS system.

The layout of distributed WAVE++ under ISIS is to run with one master process which does the problem setup, such as geometry layout, and perform any interaction with the user. Each region in WAVE++ is distributed to a separate process running under ISIS. The methods of a region are mapped to ISIS entries, which allow for concurrent execution within the ISIS process by the use of a lightweight process system. Since passing particles between regions is a method invocation, we can map this into ISIS entries which allow multiple neighboring regions to update the same region concurrently. ISIS supports locks which may be needed if simultaneous method invocations need to modify a single member data structure. In many real problems there will be more regions than processors, so there will be multiple processes running on each processor. In our current implementation, the design is asymmetric with different process bodies, one for the master process and many of the same kind for the region processes. The constructor of a region will be invoked in the master process which will in turn fire up the ISIS process for a region somewhere on the network. This region process is currently a kludge, since we are having difficulty in mapping the notion of a C++ object to an ISIS entity. The next release of ISIS (2.0) should eliminate the problems in the current ISIS interface which inhibit using ISIS directly from a C++ program. A single region will be declared as a global in each process and each public method will be indirectly called by a C function which references the global object. It is clear that at this time we are lacking language level support for distributing objects.

```
class Region {
public:
    Region();
    init(geom);
    m1();
    m2();
    m3();
};

Region R;

initialize() {
    Geometry geom; // process message args, stuff into geom
    R.init(geom);
}

method1()
{
    R.m1();
} // same for 2 & 3m

main() {
    isis_entry(initialize,...);
    isis_entry(method1,...);
    isis_entry(method2,...);
    isis_entry(method3,...);
    isis_start_main();
}
```

We are trying to stay within the confines of C++, and we are not language implementors (although we are certainly language design kibitzers), so the process of generating this code is done by hand. Fortunately, the object structure which resulted in sequential Wave++ maps well into this form. In the long term, we will need to create an distributed object class where the master's version of an object is simply a handle to the object which resides on the worker process.

## Scientific Programming with C++:

We have discussed several issues with respect to integrating concurrent programming models into the sequential model presented by C++. In the process of developing this and other scientific codes we have encountered some fundamental limitations which hamper the acceptance of C++ by the scientific programming community. Specific issues include optimization, storage management, and the implementation of aggregate arithmetic types (Matrices). We expect that the resolution of these issues will require changes to the language, and that proposed extensions to C++, such as parameterized types, could significantly further the acceptance of C++ as a scientific programming language.

### Optimization

The use of class objects to represent individual particles results in a completely scalar calculation of the particle motion. Particles are handled very efficiently in a vectorized fashion by the Fortran WAVE code. In order for WAVE++ to favorably compete with WAVE, some amount of vectorization and optimization must be regained in the object oriented implementation. Currently vectorization over arrays of objects is difficult, as object methods tend to hide the underlying vectors spread across a linear array of objects. Additionally, the space-wise encapsulation of data members within an object spreads out components over strides large enough to impede some vector engines. Similarly, machines with caches will suffer lower cache hit rates. Ideally, we would like to stick with the C++ view of an array of objects, but have the underlying object data stored with one contiguous vector for each data member. The current semantics of the `const` modifier (which, with some help from inlining, is the only available channel for communicating data dependency information to the compiler) are not sufficiently strong to enable even the best of compilers to successfully perform some simple procedural integration and loop manipulation which would otherwise enable vectorization.

### Storage Management

Vectorization is but one part of the performance issue. Because of the enormous size of problems to be solved, efficient management of the data's memory store is also a critical factor. We discuss below some significant problems encountered in developing aggregate arithmetic types with respect to data storage and unnecessary data duplication. Problems currently solved in Fortran on our Crays often utilize the full memory store. As these are non-virtual memory machines, it is essential that C++ implementations do not waste the store gratuitously, or they will never be able to handle sufficiently large problems to be of interest.

### Aggregate Arithmetic Types in C++

The overloading of operators as object methods is one of the great attractions of C++ for the community of scientific programmers. A primary goal of a matrix/linear algebra package is to enable programmers with strong mathematical and physics backgrounds, but cursory programming skills, to develop algorithms by focusing on the science rather than on implementation and coding details (the goal of all C++ class library builders). The primary constraint in this context is to develop a set of tools that are as dependably efficient as they are notationally familiar. However, anyone who has made a concerted attempt to implement a usable matrix/vector package has no doubt faced a series of difficulties and disappointments with the current C++ language. Bjarne Stroustrup himself has stated [Stroustrup 1988] that he couldn't think of a worse undertaking for a first foray into C++ programming. To our knowledge, there has been no substantial published discussion of the class of problems that arise in this context [Lea 1989], although much of it has become general lore.



Consider an initial naive implementation of a Matrix class. We'll start out with only a simple row-major, dense representation (Warning: not recommended for use in the home):

```
class Matrix {
private:
    int      r,c;
    double*  data;

public:
    Matrix(int m, int n)
        :r(m), c(n), data(new double[m*n]) {}
    Matrix(Matrix&);
    ~Matrix() { delete data; };
    int      rows() const { return r; };
    int      cols() const { return c; };
    double&  operator () (int i, int j)
        { return data[i*c+j]; };
    double   elem(int i, int j) const
        { return data[i*c+j]; };
    Matrix   operator + (Matrix&) const;
    Matrix&  operator = (Matrix&);
};

Matrix::Matrix(Matrix& m) {
    r = m.rows(); c = m.cols();
    data = new double[r*c];
    for(int i = 0; i < rows(); i++)
        for(int j = 0; j < cols(); j++)
            (*this)(i,j) = m(i,j);
}

Matrix Matrix::operator + (Matrix& m) const {
    ensure_conformance(*this, m);
    Matrix t(rows(), cols());
    for(int i = 0; i < rows(); i++)
        for(int j = 0; j < cols(); j++)
            t(i,j) = elem(i,j) + m.elem(i,j);
    return t;
}

Matrix& Matrix::operator = (Matrix& m) {
    if (&m == this) return *this;
    delete data;
    r = m.rows(); c = m.cols();
    data = new double[r*c];
    for(int i = 0; i < rows(); i++)
        for(int j = 0; j < cols(); j++)
            (*this)(i,j) = m(i,j);
    return *this;
}
```

Now consider the following code fragment:

```
Matrix a(M,N), b(M,N), c(M,N), d(M,N);
// assign values to a, b, and c here ...
d = a + b + c;
```

Internally C++ converts the last statement above into the following sequence of statements (or something similar):

```
Matrix temp1 = a.operator+(b);  
Matrix temp2 = temp1.operator+(c);  
d.operator=(temp2);
```

Note that two temporaries have been created by the compiler, and that each invocation of `Matrix::Matrix (Matrix&)` or `Matrix::operator=` involves copying the entire data array from the source to the destination `Matrix`. For large matrices, the space requirements for the two temporaries may well exceed the available memory. Instrumenting and testing this example under both the AT&T 2.0 and GNU 1.37 compilers indicates that three copies of the data array are performed, two by the `Matrix (Matrix&)` constructor and one by the assignment operator. As one could hand code the expression above with no copying of data or temporary creation, these copies and temporaries are unnecessary. As noted previously, this wasteful management of memory resources can make all the difference between a usable and an unusable scientific code.

The problem faced with temporary management is analogous to temporary register management in the compilation of expressions of primitive types such as `Real` and `Int`. An optimizing compiler knows when it can recycle a temporary register and generates the appropriate code. For non-primitive types the compiler does not perform this storage management function, thus we must mimic this function at run time. We have identified several techniques for implementing run-time temporary control for aggregate objects (the methods we are describing are applicable to any object implementing a pointer to heap-allocated store). The first technique is for each object to maintain internal state reflecting whether this instance is bound or temporary. This state is accessed and modified by the constructors, assignment operators, and a handful of manipulator and accessor functions. A second technique is to implement a helper temporary class, so that an object's type can supply the necessary state information. This technique can be expanded to include a complete run-time expression evaluator for each aggregate type. A third technique is to fall back on reference counting and copy-on-write semantics for the object's store, thereby avoiding the need to know anything of an object's temporary status. A fourth and less desirable technique is to circumvent the entire issue by avoiding constructive operators and function calls completely, relying on a procedural implementation much like that one would use in C. As there are many pitfalls to these techniques, we have provided a more detailed synopsis of the problem of temporary control in an appendix.

The lesson to be learned is that it is very difficult to implement aggregate arithmetic types in C++. We hesitate to place our finger on exactly *why* it is so difficult, but the problem seems to involve the inability to specify sufficient semantic information about a given object type to the compiler.

## Squeezing Performance out of Inheritance Hierarchies

In an implementation of a matrix package, the class `Matrix` would be an *abstract base class* [Lea 1990], defining only the semantics of a matrix and the generic interface to a `Matrix` object through the use of pure virtual member functions. All implementation details would be reserved for various publicly derived types (such as `DiagMat` and `SymmetricMat`), frequently implementing the virtual methods as inline code. The code fragment below will function properly for arguments of all types derived from `Matrix`, but will require three virtual function invocations on each pass through the inner loop.

```

void add(const Matrix& a, const Matrix& b, Matrix& result) {
    ensure_conformance(a, b, r);
    for( int i = 0; i < r.rows(); i++ )
        for( int j = 0; j < r.cols(); j++ )
            r(i,j) = a(i,j) + b(i,j);
}

```

These invocations are far more costly than the actual addition of elements. It is desirable to have a version of this `add()` routine for all possible combinations of subtypes of class `Matrix`. These versions of `add()` are identical except for the types of the formal parameters and the use of the corresponding explicit subtype method invocations. This permits the compiler to do inline expansions of what would otherwise be virtual function calls, and to perform the usual optimizations.

At present, the best one can do is to generate all the necessary or appropriate function definitions by hand. Intelligent editors and awk or perl scripts can be trained to do the source code generation from a template. However, for a three-parameter function like `add()` above, and with ten matrix subtypes (not an unreasonable number), there are one thousand possible combinations of parameter types (an unreasonable number).

We need a mechanism for folding function definitions with inheritance hierarchies, which we believe goes beyond the capabilities of parameterized function definitions. Such a facility would allow the programmer to specify a prototypical function or method based upon the semantics and interface of an abstract base class, and the compiler would assume responsibility for generating more refined and integrated instances of the method whenever more highly refined (wrt derivation) argument lists are known at compile time. In these proceedings Douglas Lea proposes a set of "customization" extensions to the C++ language to support such a notion of inheritance-based parameterization.

## Parameterized Types

Adaptive grids, kd trees, quadrees, and octrees are all parameterized types which we could use immediately in our code. The parameterization of arithmetic and algebraic *classes* such as matrices and vectors would greatly simplify future work in WAVE++. Beyond the obvious utility of matrices of integral, floating point, and complex elements implemented via parameterization lies the ability to *nest* parameterization, e.g. matrices of matrices. The representation of sparse-block or block diagonal systems as matrices where each element is itself a matrix of floating point values would simplify the implementation and parallelization of block parallel algorithms for solving the electromagnetic field equations, especially for implicit particle simulations.

## Conclusions

From our experiences with the development of WAVE++ and basic arithmetic and algebraic classes, we have found that C++ provides a useful and powerful paradigm for building modular and easily extended physics codes. Data abstraction techniques can be used to closely, but not perfectly, match the mathematical representations to the physical objects and also provides a nice paradigm for writing parallel codes. The limitations of C++ that we have encountered involve difficulties in providing the proper information to a compiler for significant optimization and vectorization, and the lack of a good syntax for expressing parallelism and referencing distributed computing objects.

## Appendix

### Temporary Management with Aggregate Types

The critical issues in temporary management are unnecessary data copying and data store utilization. By implementing aggregate types which allocate storage from the heap one can, under many circumstances, replace element-wise copying of data with an exchange of storage block references. Using this technique, temporaries become nothing more than convenient short-term stewards for storage blocks of data generated and passed around during expression evaluation. The problem becomes one of developing a sufficient semantics of temporary aggregate objects that can be implemented under the constraints of the current language.

To paraphrase the AT&T C++ Language System Reference Manual [A.T.&T. 1989], there are only two things which may be done with a temporary object once constructed. Its value can be fetched once for use in another expression, in which case the temporary may be destroyed immediately; or the temporary may be bound to a reference, in which case it cannot be destroyed until the reference goes out of scope. Any time an object is created or assigned from another like object which can be identified as a temporary, it is possible to exchange "ownership" of the data store from the temporary to the target object, providing that the possibility of binding a temporary to a reference is eliminated. All that is needed is a mechanism for determining whether an object is bound or temporary. This may be accomplished through manipulation of an object's internal state or through type information. Since temporaries occur as the result of expression evaluation, it is sufficient to ensure that all methods/functions returning aggregate objects by value set the state (either by manipulating internal state or by typing) of the returned value to indicate temporary status.

There are several techniques whereby the determination of bound state and the transfer of data storage can be implemented. One technique is to include some state in the object itself describing who owns the data storage and whether the object is temporary or bound. This state can be manipulated by the various constructors and assignment operators, as well as some carefully designed manipulator methods. While relatively easy to implement, the drawbacks of this method are rather severe. In order to ensure that a temporary is never bound to a reference, it suffices to follow two cardinal rules: always pass into functions by value (except under special circumstances), and never bind an expression or a function return value to a reference explicitly. This implies that function arguments must be passed by value, which is reasonable as long as the actual parameters are always temporaries. This is most certainly an undesirable misfeature, and a major performance penalty is imposed. (This problem may be partially circumvented by combining this technique with the indirected approach discussed shortly.) Also, it inevitably imposes upon the end user the use of certain awkward constructs in order to manipulate the state information. For example, consider the following function which returns a Matrix by value using this technique:

```
Matrix IdentMatrix(int n) {
    Matrix t(n,n);
    for( int i = 0; i < n; i++ )
        t(i,i) = 1.0;
    t.release_data(); // mark storage for release
    return t;
}
```

It is necessary to mark the storage held by the local object `t` as available to be stolen by the constructor of the return value. In fact, `Matrix::release_data()` must mark the data



storage as twice-exchangeable: once to pass the storage from the local variable to the temporary return value, and once again to pass from the temporary return value to whatever the next or final destination may be. For this reason, it is necessary to count the passes through various constructors and assignment operators to determine proper bound state.

Herein lies a serious problem of this technique. C++ compilers may also occasionally take various short-cuts. Although they frequently create temporaries, the number of these temporaries can vary from compiler to compiler and optimized compilations may even eliminate the construction of useless temporaries, under the implicit assumption that constructors perform no useful computation by side effect. There is no way to consistently ensure how many constructors are invoked in passing out of a function returning an aggregate by value to a constructor of a like typed destination. This uncertainty renders it difficult ensure that a bound object *always* owns its data (except when explicitly released) and that a temporary object never thinks itself bound. Although we have succeeded in using a matrix implementation similar to what we describe here in a large code, it was at the expense of banishing the `X x = f()` syntactic form from our vocabulary!

A second possible solution to the problem of temporary control is that of using a special helping class to hold temporary object values. By carefully avoiding the binding of an instance of the temporary helping class to a reference, we can freely treat the ownership and contents of a temporary object as we wish. In the context of our Matrix example, we can implement our matrices by creating a `tmpMatrix` class in parallel with class `Matrix`. We ensure that *all* constructive functions and methods return a `tmpMatrix` object, and that a `tmpMatrix` is never bound. By providing `Matrix::Matrix(tmpMatrix&)` and `Matrix::operator=(tmpMatrix&)` methods which always steal the data and data ownership from the `tmpMatrix`, we can ensure that all bound Matrices own their storage.

This technique does have some of the same notational convention problems as the first proposed technique, but it is not susceptible to the problems of constructor elision encountered with the earlier technique. In fact, when combined with the indirected reference-counting scheme discussed below, many of the notational inconveniences can be avoided. Furthermore, the code can be structured in such a way as to always guarantee correctness under all constructor and expression syntactic variations save one - explicitly declaring a bound `tmpMatrix`. Failure to follow certain cliches may introduce some runtime inefficiencies, but the code will always function correctly.

A third possible solution to the problem of temporary control and excessive copying is to implement the object class as an "intelligent pointer" to a reference counting representation class. Construction and assignment of the object class can simply pass around a handle to a representation object which keeps track of all live references and implements copy-on-write semantics. In fact, this technique may be used to advantage when combined with either the previous techniques.

One significant problem, however, is that the language specification for temporary destruction requires only that a temporary be destroyed before control exits the most closely encompassing scope. This means that after assigning from a temporary to a bound object, a compiler may keep the temporary and its associated reference to the underlying representation object alive long enough to force an otherwise unnecessary copy-on-write. Additionally, reference counting and the associated indirection can carry a significant performance costs. For small objects and on machines not very adept at indirection, this cost may well be prohibitive. However, on modern day RISC architectures and with large aggregate objects (our work often involves  $100 \times 100$  or  $10,000 \times 10,000$  matrices), the gains in copy avoidance and reduced storage requirements can far outweigh the costs of reference counting and indirection.

Of course, there is a fourth technique available, which is to simply avoid the use of *any* constructive functional forms. Rather than utilizing operators, one can employ the sort of procedural technique that would ordinarily be used in C, passing in by reference the operands and a pre-constructed object to receive the results of the procedure call. Although occasionally appropriate, this technique violates our sense of what C++ and object oriented design should allow us to do with aggregate arithmetic types.

A root of the difficulties in implementing aggregate arithmetic types, as Doug Lea has pointed out [Lea 1988; Lea 1989], is that C++ makes no assumptions about the semantics of overloaded operators. This renders any higher-level compile-time optimization of expressions containing constructive operator invocations impossible. It also compels one to consider the possibility of implementing expression evaluation and optimization at runtime, by use of expression-constructing object operator methods. Done on a class-by-class basis, knowledge of the full semantics of an arithmetic type can be exploited. One particular advantage in the Matrix context is the exploitation of the commutativity of matrix multiplication to reduce the operation count of multi-factor products of non-square matrices.

## References

- A.T.& T. C++ Language System Reference Manual. 1989.
- B. N. Bershad, E. D. Lazowska, H. M. Levy and D. B. Wagner, An Open Environment for Building Parallel Programming Systems, *SIGPLAN PPEALS Conference 1988*, 1988.
- K. Birman and T. Joseph, Exploiting virtual synchrony in distributed systems, *Proc. 11th ACM Symposium on Operating Systems Principle*, 1987.
- K. Birman, R. Cooper, T. Joseph, K. Kane and F. Schmuck, *The ISIS Systems Manual*, Version 1.2, 1989.
- L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys*, vol. 17 no. 4, 1985.
- D. W. Forslund, Plasma Simulation Techniques, *Space Science Reviews*, 1985.
- D. Lea, Customization in C++, *USENIX C++ Conference Proceedings*, 1990.
- D. Lea, Lecture on Embedded Languages in C++, July 1989.
- D. Lea, Users Guide to the Gnu C++ class library, Free Software Foundation, 1988.
- B. Stroustrup, Private Communication, Oct 1988.
- B. Stroustrup, Parameterized Types of C++, *USENIX Computing Systems*, vol 2, no. 1, 1989.
- M. Tiemann, "Wrappers:" Solving the RPC Problem in GNU C++, *USENIX C++ Conference Proceedings*, 1988.

# The Conduit: a Communication Abstraction in C++<sup>1</sup>

Jonathan M. Zweig

Ralph E. Johnson

Department of Computer Science

University of Illinois at Urbana-Champaign

1304 W. Springfield Ave.

Urbana, IL 61801 USA

Electronic Mail:

zweig@cs.uiuc.edu, johnson@cs.uiuc.edu

## Abstract

This paper describes the *Conduit*, a new abstraction for bidirectional communication that is being used to implement a family of network protocols as part of the *Choices* operating system project [1]. The use of C++ makes bit- and byte-level-manipulation easy, at the same time allowing the programmer to exploit object-oriented features such as type-checking, inheritance and polymorphism. The structure imposed on the system by the class hierarchy makes the code easier to understand and permits reuse of both code and interface. The result is a flexible, extensible system that can easily be tuned to provide optimum performance. Certain shortcomings of C++ that are encountered in implementing a communications subsystem are also discussed.

## Introduction

This paper describes the *Conduit*, a new abstraction for bidirectional communication that is being used to implement a family of network protocols as part of the *Choices* operating system project [1]. The main goal of the *Choices* project is to investigate the benefits of object-oriented programming by implementing an object library from which particular operating systems can be assembled. This library provides portable operating systems by encapsulating machine dependencies.

The communication subsystem was built "from the ground up" rather than reimplementing an existing subsystem using the object-oriented methodology. The *Conduit* abstraction was developed as the basis for this system. Since the existing body of knowledge about communication software includes a number of very good abstractions for communication, it was inevitable that some of these ideas would be reused in our system. Thus, the *Conduit* bears architectural similarity to some of these existing systems.

A typical application program's interaction with its operating system's communication subsystem will consist of read and write calls along with control operations for manipulating the network services being used. Thus, the design of the communication subsystem must support this "get/put" style of interaction. One possibility would be to structure the entire system as a group of connected modules, with each module containing a buffer and a task that gets information from a lower layer module, processes it, and places it in its buffer, ready for use by higher layers. Data put into a module by a layer would be processed by this task periodically. This approach

---

<sup>1</sup>This work was supported by a Ph.D. fellowship from the AT&T Bell Laboratories Foundation, AT&T METRONET grant number 1-5-37411 and NSF grant number CDA 8722836-1-5-30035 (Tapestry).

could reduce the need for synchronization inside a module since there would only be a single process manipulating the internal data structures at any given time.

Conduits, on the other hand, use a "put/put" style of interaction in which, in both directions, a Conduit with data ready for another Conduit inserts it into that Conduit. This allows some Conduits to be passive; they take no action except when messages are inserted into them. Others can have daemon processes that transport messages through (part of) the protocol stack.

Each Conduit has two ends into which messages may be inserted. The *top* is connected to the application side of a conversation, the *bottom* is connected to the network side. This parallels the up/down notion found in the OSI Reference Model ([2], pp. 14-27). The basic interaction between Conduits is the insertion of messages, although subclass-specific interfaces add operations such as opening and closing virtual-circuits, managing sliding-windows and so forth. A Conduit may respond to having a message inserted by inserting a message into the next Conduit, which may do likewise. The calls to insert-operations can be nested, finally popping back out when the message is dealt with in some way, such as being transmitted or inserted into a buffer. We believe that this approach will minimize overhead due to context-switching and simplify the system, resulting in better performance. A thorough discussion of performance issues, however, is beyond the scope of this paper.

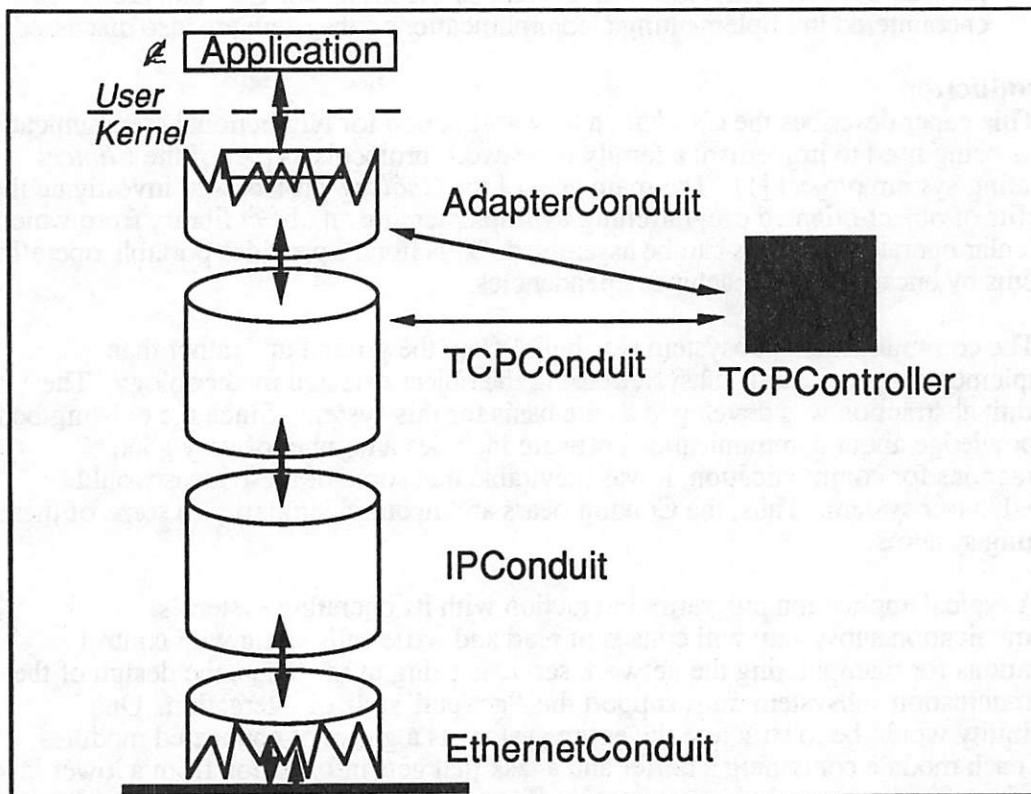


Figure 1.

#### An Example: TCP/IP

As an illustration, consider an implementation of the DoD Transmission Control Protocol/Internet Protocol (TCP/IP) suite, depicted in Figure 1. The application-



program interacts with an Adapter-Conduit, which takes write requests from the application and packages them as Conduit-messages and buffers incoming Conduit-messages from TCP so that it can serve the application's read requests<sup>2</sup>. The messages inserted into the TCP Conduit contain only the application's data. The TCP-Conduit must add to each of them a TCP-header that will carry information such as TCP-checksum and sequence-number. Since this information is contained in a connection descriptor maintained by the TCP Conduit, some addressing information must accompany each message being inserted into its top which specifies the particular session the message is being sent on. The TCP Conduit will then encapsulate each message in an IP message which similarly does not yet have an IP header<sup>3</sup>. This message is then passed to the IP Conduit, which determines the appropriate local data-link level address (in our example, Ethernet is used) to which to send the datagram, and also fills in the IP-checksum and other fields in the IP header. It thus encapsulates the IP-message inside an Ethernet-message, which is passed into the Ethernet Conduit to fill in the address information. The Ethernet Conduit translates the Ethernet message into a set of pointers and lengths and passes them to the DMA controller which is able to copy all the octets forming the packet into the Ethernet interface card's local memory for transmission over the Ethernet.

Notice that the messages flowing between two Conduits always have the type expected by the lower of the two. This way, for example, the IP Conduit does not need to have any information about the message-types used by each of the several types of Conduits that could be attached to its top. Its clients translate their own message types into IP-messages before sending them down, and expect to receive IP-messages coming back up which need to be translated into their own types. This convention also facilitates the direct transmission of information that would otherwise require additional processing to recover. For example, each Ethernet packet that is transmitted includes a field indicating the protocol module that generated the packet. Since the IP Conduit knows it is connected to an Ethernet Conduit, and must therefore also know its Ethernet protocol number, it can simply mark this field in the header of each Ethernet message it inserts into the Ethernet Conduit. Otherwise, the Ethernet Conduit would need to consult its table of protocol-number to Conduit mappings (which it uses to demultiplex incoming packets) to map the sending Conduit to its protocol-number, which would be significantly more costly. This is not, however, a serious violation of the division between protocol levels, such as would occur if one Conduit were to directly manipulate internal data-structures belonging to the Conduit below it. Since each protocol relies on the capabilities of the protocol below, it must understand certain aspects of that protocol (such as the upper-level protocol identifier in each Ethernet packet) and leave other aspects of the lower-level protocol processing (such as the CRC) to the module that implements it. The fact that TCP and IP do not separate cleanly into two layers is well known.

Messages arriving from the network are delivered as an array of octets by a low-level device driver to the Ethernet Conduit for processing. It creates an Ethernet message with pointers into the appropriate parts of the packet and delivers the message to the IP Conduit (whose Ethernet protocol type number was previously registered with the Ethernet Conduit). The IP Conduit will then "unencapsulate" the IP-packet contained as data in the Ethernet message by creating a new IP message whose header

---

<sup>2</sup> The section "Conduits in Detail" below gives a more detailed explanation of how this works.

<sup>3</sup> This is not strictly true (see below) but is a useful way of envisioning what ought to be going on.

points to the beginning of the Ethernet-data, and whose data points to the octets following the IP header.

The IP Conduit then decodes the IP-message's protocol field and demultiplexes the message to the TCP Conduit which, after unencapsulating the TCP-data in the IP-message and checking that there is sufficient space in the Telnet Conduit's receive-buffer, updates the sequence-number of the latest data received and other internal TCP data structures. It then inserts the TCP-message into the Telnet Conduit which stores a pointer to the message until the data is called for by a subsequent application read-request. After the data have been read by the application, the message may be deleted.

### Conduits in Detail

Figure 2 gives a slightly abbreviated definition of the class Conduit. The basic operations on Conduits fall into three categories; connecting Conduits together, disconnecting them, and inserting a message into one of the ends of a Conduit. Note that here `Return_Code` is an enumerated type consisting of OK, ERROR, and so forth. `ProtocolID` is another enumerated type which assigns to each of the subclasses of Conduit a unique identifier.

The operation `connectTo` instructs the Conduit to attempt to connect to the specified Conduit `c` by invoking `connectFromAbove` on it, specifying itself and its own type (protocol-ID) as arguments. If, after inspecting the protocol-ID of the first Conduit, the second wishes to connect, it will invoke the `connectFromBelow` operation on the first, specifying itself and its own protocol-ID. This "double-dispatching" ensures that each Conduit is given the opportunity to inspect the protocol-ID of the other to check that the requested connection makes sense (see "Getting Connected" below). The reason for specifying a pointer to the second Conduit in `connectFromBelow` is to allow the possibility for multiple simultaneous invocations of `connectTo` to be pending on a single Conduit at the same time. When a call to `connectTo` returns OK, it is known that both Conduits have accepted the connection and initialized their internal state to reflect the fact.

```
class Conduit : public SystemObject {
protected:
public:
    Conduit();
    virtual ~Conduit();
    virtual Return_Code connectTo( Conduit * c );
    virtual Return_Code connectFromAbove( Conduit * c,
                                           ProtocolID proto );
    virtual Return_Code connectFromBelow( Conduit * c,
                                           ProtocolID proto );
    virtual Return_Code disconnectFrom( Conduit * c );
    virtual Return_Code disconnectFromAbove( Conduit * c );

    virtual Return_Code insertFromAbove(
                                           ConduitMessage * msg,
                                           ConduitAddress * addr );
    virtual Return_Code insertFromBelow(
                                           ConduitMessage * msg );
}
```

Figure 2.

Since there is no reason to check the type of a connected Conduit again, the disconnect operations need not double dispatch; the class just needs to provide an operation to allow an external controller to request disconnection, as well as an operation for the upper Conduit to indicate to the lower that it should disconnect.

The procedure `insertFromAbove` is used by a Conduit connected to the top of another to insert messages. Notice that this operation also takes the address of the message's destination as an argument. For example, inserting a message that does not yet have an Ethernet-header into an Ethernet-Conduit requires that an ethernet address be specified indicating where the message should be sent, while a message inserted into a TCP-Conduit needs a session identifier to indicate which virtual-circuit session it is being sent on. The `insertFromBelow` operation, on the other hand, is used for delivering messages received from the network that already contain the full address of their destination.

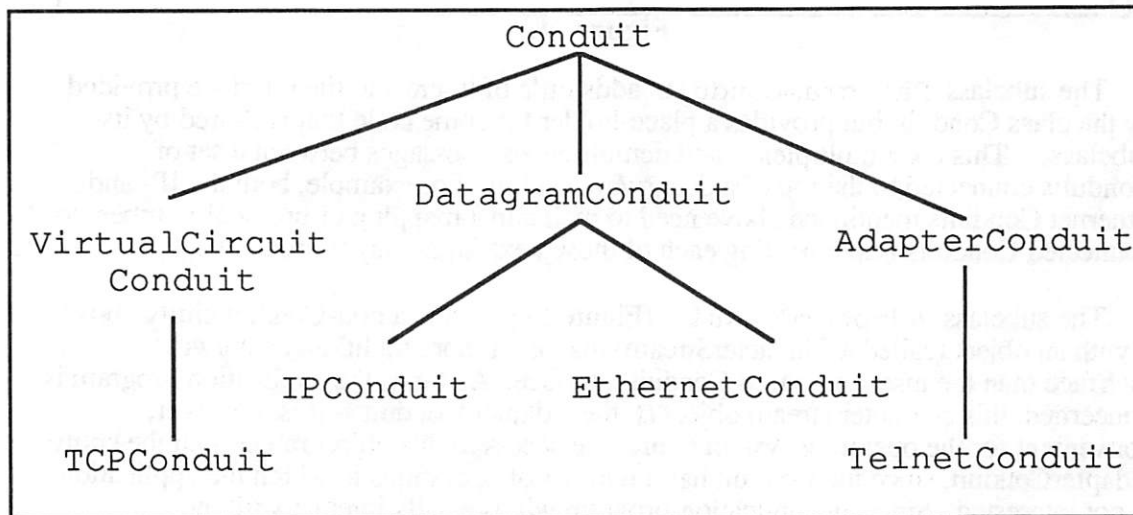


Figure 3.

As shown in Figure 3, three distinct subclasses of the class `Conduit` are used to provide a framework for structuring the concrete subclasses that implement each protocol. The subclass `VirtualCircuitConduit` imposes the semantics of virtual-circuits (i.e. expresses the contract that the order in which messages are inserted should be preserved in passing them through to the other end, and that reliable delivery of all messages should be performed), and adds the operations `open`, `close` and `abort` to create/destroy virtual circuit sessions. It also introduces the management of a sliding-window flow-control protocol by requiring that the `Conduit` opening a virtual-circuit connection specify as a parameter to the `open`-operation the capacity of its receive-buffer, and requiring that any increase in the space available in its receive-buffer (i.e. after incoming messages are read by the application) be explicitly indicated through the `expandWindow` operation (see Figure 4). In general, every octet of incoming data on a virtual-circuit reduces the space in the receive buffer by one, so a `VirtualCircuitConduit` can control data flow by advertising the receive-window to its peer. Examples of `VirtualCircuitConduits` could include TCP Conduits, Appletalk Data Stream Protocol (ADSP) Conduits [3], and ISO Class 4 Transport Protocol (TP4) Conduits. (TP4 is described in [2] pp.422-437.)



```

class VirtualCircuitConduit : public Conduit {
protected:
public:
    VirtualCircuitConduit();
    virtual ~VirtualCircuitConduit();
    virtual Return_Code open( Conduit * c,
                              int initial_window_sz,
                              ConduitAddress * dest,
                              ConduitAddress & sess );
    virtual Return_Code expandWindow(
                              ConduitAddress & sess,
                              int additional_space );
    virtual Return_Code close( ConduitAddress & sess );
    virtual Return_Code abort( ConduitAddress & sess );
}

```

**Figure 4.**

The subclass `DatagramConduit` adds little of interest to the interface provided by the class `Conduit`, but provides a place-holder for some code that is shared by its subclasses. This code multiplexes and demultiplexes messages between a set of Conduits connected to the top of a datagram Conduit. For example, both the IP- and Ethernet Conduits mentioned above need to maintain a mapping of protocol-numbers to connected Conduits implementing each of those next-higher layer protocols.

The subclass `AdapterConduit` (Figure 5) presents a non-Conduit entity above it with an object (called a `CharacterStream`) that has a more traditional read/write interface than the insertion-based Conduit interface. As far as the application program is concerned, this character stream object *is* the Adapter Conduit -- it is, however, convenient for the operating system to provide access to this object rather than the entire `AdapterConduit`, since the Conduit has a number of operations in which the application is not interested. Since an application-program will typically interact with the communication subsystem using read and write calls rather than being written in an event-driven style, it is convenient to "cap" a set of connected Conduits with an `AdapterConduit` which plays a role analogous to that of a stream-head in AT&T UNIX System V Streams® [4]. Of course, an application program that wishes to present a Conduit-based interface to the subsystem may do so. The `AdapterConduit` is merely a favor to the application programmer. The Telnet Conduit mentioned above is an `AdapterConduit` that has the additional functionality of intercepting Telnet commands buried in the data-stream before the user sees the data (rather than copying the user-data into a buffer prior to the user's read-request, it copies the data out of queued TCP-messages into application-specified buffers, eliding the Telnet commands).

The TCP Controller object mentioned in Figure 1 keeps track of the location of the TCP Conduit (and its associated helpers connected below) so that application programs can create and destroy connections with it (this is important because an `AdapterConduit` has to get connected to the TCP Conduit before being able to create virtual-circuit connections with remote systems). The reason the controller knows the location of the TCP Conduit is that it is the object responsible for creating and connecting the TCP-, IP- and Ethernet Conduits at system initialization.



## Getting Connected

The class `Conduit` defines an interface for connecting Conduits that all Conduits must provide (see Figure 2). The `connectTo` operation tells one Conduit to connect with another below it. Since this operation can be invoked on any Conduit, the topology of a set of communicating Conduits can be changed arbitrarily (in contrast to Streams which only allows pushing and popping processing modules onto or off of a single stream). Connect also allows for protocol-identification exchange since certain types of message exchange don't make sense. As an example, consider a hypothetical system in which user messages containing data but no control information are inserted into a Conduit that adds a header to each message and inserts the message into another that transmits the packet over the network. Connecting the user's Conduit directly to the network interface Conduit would result in nonsensical packets being transmitted. The mechanism provided is sufficient to ensure that Conduits that rely on properties only possessed by certain other Conduits can ensure at connect-time that the connection is appropriate. A more powerful mechanism could be based on "first class" classes. Optimally, it should be possible for a Conduit to check that a Conduit to which it wishes to connect is a member of the least specific subclass of Conduit possible. While it is possible to accomplish this by careful administration of the protocol-ID numbers, it would be much more convenient and flexible if classes were objects that allowed a Conduit to check whether another Conduit's class was a subclass of a given class at run-time. See "Shortcomings of C++" below.

A given Conduit can be connected to more than one Conduit on either or both of its ends, so a separate class for objects that do multiplexing/demultiplexing is not required. A Conduit is merely an object capable of delivering messages on behalf of another connected Conduit, so whether or not a particular Conduit uses a mechanism to differentiate between several users of its services is up to the programmer who wrote that Conduit's code. The architecture does not restrict the topology of each Conduit.

The notion of a set of Conduits connected to each other is not the same as the notion of a connection or session in the sense of a virtual circuit or stream. Connecting two Conduits merely means that they are prepared to exchange messages between each other. This usually involves maintaining a pointer to the Conduit(s) to which one is connected, although the exact semantics of the `connect` operations may vary from class to class. There can, for example, be several TCP connections between a single application-layer Conduit and a single TCP Conduit.

```
class AdapterConduit : public Conduit {
protected:
    ConduitCharacterStream * _inputOutputStream;
public:
    AdapterConduit();
    virtual ~AdapterConduit();
    virtual ConduitCharacterStream * stream();
}
```

Figure 5.

```

class ConduitMessage {
protected:
    int      _control;          // subclass-specific
    semantics
        octet * _header;
        int      _headerlength;
        octet * _data;
        int      _datalength;
        ConduitMessage * _link; // ptr to msg being carried
        ConduitMessage * _hldr; // ptr to msg carrying me
        char      _status;      // stores delete-flags
public:
    ConduitMessage();
    ConduitMessage( int h_size, int d_size );
    ConduitMessage( int h_size, int d_size,
                    ConduitMessage * payload );
    virtual ~ConduitMessage();
    virtual int      length();          // overall length
    virtual octet * header();
    virtual int      headerLength();
    virtual octet * data();
    virtual int      dataLength();
    virtual ConduitMessage * link();
    virtual ConduitMessage * holder();
    virtual int      control();
    virtual void setHeader( int new_h_len,
                           octet * new_h );
    virtual void setData( int new_h_len, octet * new_h );
    virtual void setLink( ConduitMessage * m );
    virtual void pleaseDeleteHeader();
    virtual void pleaseDeleteData();
    virtual void pleaseDeletePayload();
    virtual void pleaseDontDeleteHeader();
    virtual void pleaseDontDeleteData();
    virtual void pleaseDontDeletePayload();
    virtual void del();
}

```

Figure 6.

### Getting Through

The class Conduit also defines a basic interface for message exchange between connected Conduits. As mentioned above, sending messages from the application side of a conversation toward the network side is accomplished with the `insertFromAbove` operation, and messages coming the other direction are accepted by each Conduit's `insertFromBelow` operation. In our system, messages are structured objects consisting of both data and control-information. Figure 6 shows the definition of the class `ConduitMessage`.

Although messages are objects, they can always be represented as a string of octets so that they can be transmitted over a network. A lower level message, such as an IP Message, may contain a higher level message, such as a TCP Message to reflect the way

that an IP packet can be used to carry a TCP packet as its payload. Thus an object of class IP-Message has a member variable (`_link`) that points to a TCP Message (it can, in fact, point to any other type of message -- there is no point differentiating between, say, IP messages that carry UDP messages and ones that carry TCP messages).

Since any message will be represented as a string of octets when it is being transmitted over the network, each message has a way of retrieving these octets (regardless of their semantics with respect to the particular message's type). Each `ConduitMessage` provides a pointer to its header (if any), data (if any) and optionally a link to another message whose octets will follow its own when transmitted. Notice that in general there is no reason to distinguish between payload and data -- in fact the payload typically *is* the data. It turns out that allowing a message either to point to its data directly as an array of bytes, or indirectly through a pointer to another message, makes the encapsulation/unencapsulation referred to above easier. For example, as described above, an IP-message might have a pointer to 20 octets of header, no data, and a link to a TCP-message consisting of a TCP header and data belonging to the application that created the message. The link permits arbitrary encapsulation of messages within other messages and can also be used for protocols that add trailers to existing messages.

If a message P is the payload of a message M, the back-pointer `_holder` of P points to M, which helps to properly manage memory. It is used by the operation `del`, which follows the linked-list backwards through the `_holder` pointers until it finds the head of the list and deletes that `ConduitMessage`. That way, any of the messages in the linked list can be used as the basis for deleting the entire message. This is necessary because, as pointed out before, an application Conduit may receive a pointer to a TCP message that is actually encapsulated in both an IP- and an Ethernet message -- in order to free up all the memory associated with the message, the TCP message must have a way of deleting the other two. The convention that is used for managing memory associated with messages is that the message at the head of the linked list (i.e. the one whose `_holder` pointer is null) should be deleted, and its destructor will delete any storage associated with it as well as deleting any message to which it is linked. Thus, destructors will get called recursively down the list until the final message (linked to nothing) is deleted. The fact that a message needs to know about the message in which it is encapsulated is partially to be blamed on the memory-management capabilities provided by C++.

The instance variable `_status` specifies which of the items being pointed to should be deleted by the message's destructor. It contains flags that can be set and cleared by the procedures `pleaseDeleteHeader`, etc. These flags provide a uniform mechanism for memory management, given the policy that whenever a message is accepted by a Conduit, it will be deleted when it has been delivered (or discarded). A Conduit may protect the contents of a message from being deleted by setting these control flags to retain responsibility for deleting those data. For example, if a Conduit may need to retransmit a message, it can encapsulate it in another message set not to delete it when transmitting it, deleting the message itself later on. Another case in which parts of a message ought not be deleted is when a single call to `new` is used to allocate an array of octets for an entire message, whose pointers are set up to view this single array as a header followed by data -- calling `delete` on the data-pointer which points into the middle of the array would be an error.



The `control` field lets each message pass along a small piece of control information along with its data. Since most subclasses of `Conduit` will define operations for manipulating data traffic, this feature is not strictly necessary -- the same effect could be achieved by invoking control-operations on a `Conduit`. But since the control field is part of a message rather than part of the interaction with a specific `Conduit`, it is sometimes preferable to convey information with it. As an example, consider a marker that indicates that urgent data is contained in a message -- it is more natural to tag the message itself ("in-band" signaling) rather than indicating urgent data by invoking an operation on the message's recipient ("out-of-band") which would require special handling of the message inside each `Conduit` through which it passes.

### Getting There

In order to connect `Conduits` and exchange messages it is necessary to have a set of objects for manipulating the data contained in the messages. Since C++ allows the programmer to use the bit- and byte-level manipulations found in C, implementing the low-level aspects of protocol processing is straightforward. In our system, a set of classes called `Words` manages machine byte-order to network byte-order translations conveniently. The key observation that lead to the interface to the class `Word` was that the programmer typically assigns values that are in network byte order to/from a pointer into an array of octets that forms a datagram, whereas host byte order values are typically stored in integers. Thus the class `Word` overloads operator `=` to treat integers as representing host byte-order quantities, and pointers to octets (unsigned chars) as network byte order quantities. Similarly, two procedures are provided to access the quantity stored in a `Word` -- given a word `w`, `w.val()` returns the host byte order value of `w` in an int of the appropriate length, and `w.cpInto( p )` will copy the value from `w` to the appropriate number of octets pointed to by `p` in network byte-order. By using inline function expansion, no overhead at all is encountered on machines whose local byte order matches network byte order ("Big Endian"). Having written both the byte-order reversing versions of all the operations for the `Word` classes as well as the "do nothing" versions, then, the entire system can be converted from using one local byte ordering to another by changing one symbolic link for a source header file and recompiling the system.

Other examples of low-level manipulations are accessing bit-structured fields within message headers and translating between arbitrary values (short and long ints) and fixed number-of-bit fields. Here again, inline function expansion can produce highly efficient programs from readable, portable code. All fields inside `ConduitMessages` are accessed using member-functions, which can typically be inlined. This is important in the case where virtual functions are required in order to provide the correct value. Consider a case in which TCP-messages may either be encapsulated inside IP-messages or in some other network-layer message type (such as AppleTalk). Since a TCP message's destination address is recorded as part of its encapsulating message, a different operation would be required to manipulate the destination-address field of a specially-encapsulated TCP message than for ordinary IP-encapsulated TCP messages. Of course, in a system in which ordinary encapsulation can be assumed, the `DestinationAddress` procedure can be defined to be an inline fetch to optimize performance.

### Getting There In Style

The system takes advantage of code reuse through inheritance. The underlying framework for protocol processing modules makes it easy for the programmer to create `Conduits` for processing new protocols or "tuned" versions of existing `Conduits`. The structure imposed on the system by having a hierarchical set of classes can make the



communication subsystem more understandable, and facilitates implementing new protocol-processing modules by reusing code for a similar existing protocol and changing only what is needed to realize the new protocol. Further, it is possible to structure the objects that implement the communication subsystem in a class hierarchy that parallels the conceptual similarities between the protocols themselves, and then to connect the protocol processing modules in a manner that parallels the organization of the protocol-stack(s) being realized. Again, this allows a programmer familiar with a set of protocols to understand a particular implementation more easily.

The *Choices* architecture provides an object-oriented kernel, and it is possible to alter the communications subsystem at run time by creating new Conduits and connecting them together. It is not necessary to preconfigure the kernel to include a specific amount of resources for the communication subsystem. Notice that, since all Conduits have names and are accessible to other kernel objects through the appropriate nameserver, there is no intrinsic constraint on the topological manipulations that can take place at run time. Rather than naming each conversation and only allowing modules to be added/removed at the top of the stack (as in Streams), any Conduit in the system could change its connections with others at any time, as long as the particular (re)connections make sense.

The result is an easily tailorable and extensible system designed to facilitate protocol-processing experimentation as well as system configuration/administration.

### Shortcomings of C++

Three language features not currently available in C++ are desirable in implementing a communications subsystem: *Parameterized classes* [5] would be useful for creating objects that manipulate messages in some generic way without knowing (at the time the code for such an object is written) the type of the messages; *Delegation* [6] would be useful for implementing Conduits whose behavior changes dynamically at run time; *Classes as First-Class Objects* would facilitate making "type-of" decisions.

An example of the first problem is a Conduit that buffers messages until a certain amount of data is ready and then sends a single message containing all the data. As long as all subclasses of `ConduitMessage` provide the same mechanism for accessing their data, the actual type of messages being buffered is irrelevant. Currently, a buffer to hold the messages inside such a Conduit would have to be made a buffer of `ConduitMessages`, with the user of the buffer explicitly casting a pointer to any message fetched from the buffer to the particular subclass of `ConduitMessage` the message is assumed to be. It would be preferable not to have to "break" the type system in this way -- since each such buffer will typically only be used to store messages of a certain type, this is a natural application of parameterized classes. Another candidate for parameterization is the set of subclasses of the class `Word` which must each define their `val()` operation individually since, for example, `Word16::val()` returns a 16-bit int, and `Word32::val()` returns a 32-bit int. Precisely the same code is used to implement both classes, with the term "int32" substituted in the latter for the term "int16" in the former.

An example of the second sort of problem is encountered with a virtual-circuit-oriented protocol that must process messages differently based on whether a connection with the foreign host has been established yet, whether either end has requested the connection be closed, etc. It would be most straightforward for such a Conduit (or part of the internal structure of such a Conduit) to delegate to an object which processes

messages appropriately for the current state of the connection. When the state of the connection changes, the object to which it delegates could be changed to realize the new behavior. In fact, the system uses precisely such a setup to handle the panoply of states in which a given TCP session can be at any given time. Doing delegation "by hand" is relatively painless, however, and it would not do well to take sides in the ongoing debate about whether to add features to the language. It is evident, however, that delegation is a useful way of implementing objects whose behavior changes at run time.

Classes as first-class objects would make the run-time protocol-checking done by the connect operations both more straightforward to implement and more sensible. For example, a TCP Conduit wants to connect to a Conduit below it that implements the IP protocol -- it makes no difference whether it is IP over Ethernet, Serial-Line IP (SLIP) or something else altogether. The best way to make this check is to determine whether a Conduit to which it is asked to connect is an instance of a subclass of the class IPConduit. This is currently done by assigning protocol-identifiers so that the high-order bits specify a protocol (in the sense of IP, TCP and so forth) and the low-order bits disambiguate specific subclasses. This two-level hierarchy is obviously not general enough to implement "type-of" checks fully. If each object's class were an object, and each class-object had a pointer to its superclass, this checking could be done in a totally general way by simply following a linked list of pointers to class-objects, checking whether any of them are the class in question.

In all three cases, there are programming conventions that allow these capabilities to be exploited, but it would be cleaner if the functionality were provided directly by C++.

Also, as pointed out above, the memory-management facilities of the language can lead to code that performs counterintuitive manipulations (such as having to delete messages from the "outside in" with some associated pointer-chasing) to ensure that all memory is ultimately reclaimed. This is especially important for code such as the communication subsystem that will run in privileged mode, since there is typically no good mechanism to deal with memory leaks within the kernel of an operating system.

### **Relation to Existing Work**

It has been realized for some time that the high degree of complexity in communication software systems implies that protocols should be divided into layers that are processed by separate software modules. Many architectures for communication protocols, including Streams and the X-Kernel [7] as well as Conduits, provide explicit mechanisms for the programmer to connect these modules at runtime to provide dynamically tailorable protocol processing. This requires defining interfaces through which modules interact, in order that functionally equivalent modules can be substituted for one another. It is thus a fairly natural extension of this view of communication subsystems to use an object-oriented language such as C++ which provides mechanisms for defining these interfaces directly in the language, and allows the reuse of the code to implement the parts of the system. The *x*-Kernel describes itself as object-oriented, although it is written in C and is thus unable to take full advantage of the facilities provided by a truly object-oriented language, such as inheritance and type-checking. Although developed independently, the Conduit-based system shares many of the same motivations, and so the overall architecture is similar.

Conduits differ from Streams processing modules in that Conduits are shareable system objects. For example, two applications wishing to use TCP/IP can be connected to the same TCP Conduit, rather than each pushing its own (private) TCP processing

module onto its stream. That is, the Conduit abstraction allows different users to share protocol state information as well as just protocol processing code. Further, the abstraction makes no stipulation about the internal structure of a Conduit (whereas, for example, Streams views processing modules as containing a queue of messages belonging to the stream's owner), and so provides a less restrictive way of thinking about the communication subsystem.

## Conclusions

The Conduit abstraction is a straightforward way of thinking about message-based communication software. The interaction between Conduits can be based either on a general message-passing interface or, when the need arises, on protocol-specific interfaces provided by subclasses. Inheritance also encourages code reuse and interface regularity. By providing a truly object-oriented mechanism that allows the programmer to organize a communications subsystem which parallels the organization of the protocols being implemented, and easily to tailor that system, the Conduit abstraction is a useful framework within which to implement communication software.

## Bibliography

1. Roy Campbell, Vincent Russo, and Gary Johnston. "The Design of a Multiprocessor Operating System." In *Proceedings of the USENIX C++ Workshop*, 1987. Also Technical Report No. UIUCDCS-R-87-1388, Department of Computer Science, University of Illinois at Urbana-Champaign.
2. Andrew S. Tanenbaum. *Computer Networks*. 2nd ed. Englewood Cliffs, New Jersey, Prentice-Hall Inc., 1988.
3. G. S. Sidhu, R. F. Andrews, and A. B. Oppenheimer. *Inside Appletalk*. Reading, Massachusetts, Addison-Wesley Publishing Company, 1989.
4. D. M. Ritchie. "A Stream Input-Output System." *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, October 1984.
5. Bjarne Stroustrup. "Parameterized Types for C++." In *Proceedings of the 1988 USENIX C++ Conference*, Denver, Colorado, October 1988.
6. Henry Lieberman. "Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems." In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, September 1986.
7. N. C. Hutchinson and L. L. Peterson. "Design of the x-Kernel." In *Proceedings of the ACM SIGCOMM '88 Symposium*, Stanford, California, August 1988.
8. Bjarne Stroustrup. *The C++ Programming Language*. Reading, Massachusetts, Addison-Wesley Publishing Company, 1986.





# Writing a Gateway in C++

Preben Fisker Jensen, Peter Juhl

pfj@jt.dk, peju@jt.dk

Jutland Telephone

Denmark

## Abstract

One of the claims concerning C++ is that it is designed to be a systems programming language. If so, it should be efficient for the development of gateways. This paper describes such a development project. We have implemented a PC based gateway in C++. Called the XGate.

We will describe the important aspects of the implementation and briefly discuss our experiences using C++ for gateway development.

We conclude that C++ is suitable when one wants to write general devices drivers, e.g. network drivers. Furthermore we conclude that C++ makes it easier to translate ones conceptual system model into code.

## 1 Introduction

In this paper we will describe a general purpose multitasking kernel developed as part of this project. Secondly we will outline the system structure, showing how we implemented an event system in the gateway. Thirdly we will describe how we achieved a high degree of network independency. Finally we will draw some conclusions about using C++ for gateway writing.

The gateway described in this paper was developed at Jutland Telephone from May to November, 89. The intention was to connect an obsolete network with a network sold by Jutland Telephone. The potential customers wanted to keep their old system while migrating to the network sold by Jutland Telephone. It was therefore necessary to write a gateway connecting the two networks. For various reasons it was decided that the gateway should be

based on PC adaptor cards. This meant that we had to work under DOS.

We used the project as a pilot project for evaluating the usability of C++ in connection with gateway development. The version of C++ used was a PC-port of cfront 1.2 from Guidelines. The backend compiler was Microsoft C 5.1.

## 2 The System

A gateway usually requires multitasking and real-time response. The system we have implemented is capable of supporting 32 simultaneous sessions. It is possible to gather statistics concerning the individual sessions interactively and present the statistics on a monitor.

The system was written in such a general way that it is possible with a minimum of efforts to rewrite the gateway, supporting other networks. This objective was achieved to a large degree.

## 3 Environment

From earlier experiences with gateway writing we knew that the best model would be a process based system. In other words, we had to implement a basic process concept. This could be used for the construction of named process descriptors. The `class` concept of C++ proved to be suitable for defining the basic process type. The extension to general named process descriptors can now be done by subclassing. This means that it is quite easy to define processes with different functionality, just write a new process subclass.

Since DOS as an operating system does not

support any of the essential things necessary for processes we had to develop a suitable kernel for our purposes.

Our starting point was the code published in [Hol87]. This code implemented a basic multitasking kernel. The code was written in C and assembly language. Since the published code did not quite suit our purposes, we did some extensive rewriting. We implemented the concepts described in [BH77] in C++ and a timer class in order to provide realtime services.

### 3.1 Process Implementation

First of all we implemented the `process` class. Every process in the system must be a subclass of class `process`.

```
class process : object {
    process ( priority, stack_space ) ;
    virtual main () ;
}
```

Every subclass must have a function called `main`. When the multitasking system is started, the `main` function in the subclasses will be called.

There is no destructor associated with the `process`. In the actual system, it is not necessary to be able to kill a process.

It can now be checked at compile time that each process only uses its own variables. This eliminates a whole class of errors.

### 3.2 Monitors and Queues

The processes alone are not enough. They must be able to communicate with other processes. We chose to base our communication concept on monitors and queues. The first thing we discovered was the fact that it is not easy to use the inheritance mechanism in C++ to implement a monitor class concept. We think that the problem is that monitors are used to control dynamic behaviour while inheritance controls static behaviour. Fortunately, C++ offered an elegant solution based on dynamic class instantiation.

The implementation of monitors takes advantage of C++ automatic initiation via constructors and destructors in blocks. The idea comes from [BLL88]. The basic idea is to regard a monitor as a variable. This allows a more fine grained monitor structure than the typical approach, where we would have a `Monitor` superclass thus ensuring that all derived classes will be `Monitor` classes. In our system, a typical piece of code will look like the following:

```
{
    // this block will be protected
    // by a monitor
    Monitor Entry(sem) ;
    .....
}
```

The idea is to declare the Semaphore `sem` as a class variable, and then use it to protect critical parts of the class code. The constructor for a `Monitor` will be called each time we enter the critical section. The constructor will ensure that only one process at a time will gain access to the critical section. When the monitor variable is out of scope, the destructor will be called. The monitor will now be released and any waiting process may try to gain access to the critical section. Our monitor implementation ensures fairness between processes competing for the same critical section.

It was also possible to protect critical code by having semaphores surrounding the code. This would mean that we should write something like `sem.wait()` when we want to enter the critical section. When we want to leave the critical section, we must write `sem.signal()`. This is easy to forget. We think that the other method using constructors and destructors is more elegant and less error prone.

The actual implementation of monitors via semaphores is described in [Hoa78].

We implemented the queue concept from [Hoa78]. Queues was implemented to support an event system. Queues have two operations

- `Delay()`, delays the calling process for an unlimited amount of time.

- `Continue()`, continues any processes sleeping in the queue.

Queues are implemented via semaphores.

### 3.3 Timeouts

Finally we have implemented a timeout concept. The timeout concept was implemented in such a way that the kernel could provide real-time services. This part of the kernel only was written with gateways in mind. The kernel administers a set of timers. Timers are granted to processes by the function

```
timer* Wait_Elem_Create(restart_time,
                        category,
                        (*wakeup)(),
                        parameters)
```

A return value of `NULL` indicates that the timer could not be created. There are two possible values for the category

- `VIRTUAL`, used for polling timeout.
- `REAL-TIME`, normal time outs.

`VIRTUAL` timeouts are handled immediately in kernel mode when they occur. The `REAL-TIME` timeouts are handled by a special process which will handle the timeout when it is scheduled.

Restart time specifies the number of seconds that the timer will wait before it times out. When a timeout happens, `wakeup()` is called with `parameters` as an argument. Timers also have operations for starting and stopping themselves.

### 3.4 Did It Work ?

If the kernel should be of any use, it must be efficient (read fast). Very often this would imply that it should be written mainly in assembly language. We did not observe any efficiency loss due to C++. As a matter of fact, it was small trivial, high level changes which gave speed, not rewriting in assembly language.

Since the kernel was conceptually separated from the rest of the gateway, we made it as a separate project. We found that since the concepts were so few and clean, it was quite "easy" to test and debug the process and monitor implementations. We think, assuming agreement on interfaces, that the kernel could have been developed and tested by an independent team not involved in gateway development. This supports the claim that C++ is designed for software developed in teams.

## 4 System Overview

In this chapter, we are considering the process structure in the gateway. The gateway supports a number of virtual sessions that connects addresses in two different networks. For terminal sessions the addresses are determined at runtime, while the user defines the addresses for printer sessions before the start of the gateway.

The gateway is managed by events. When an event is detected, a corresponding action takes place somewhere else in the system. Figure 1 describes the processes involved in the system. The process structure is chosen to ensure an easy flow of events through the system. Both networks is monitored by a card process, whose task is detecting events that happens on the network hardware and pass it on to another process in the system. For every virtual session there exists two driver processes responsible for incoming events on the two networks. The card processes deliver the events through network monitors, which are the drivers interface to the network. A monitoring process is responsible for showing the gateway-activity on a screen. The class `admin` is responsible for global information, and will be described in further detail in section 5.

### 4.1 Event Structure

Both drivers are listening for the following events:

- `IN-DISCONNECTION`

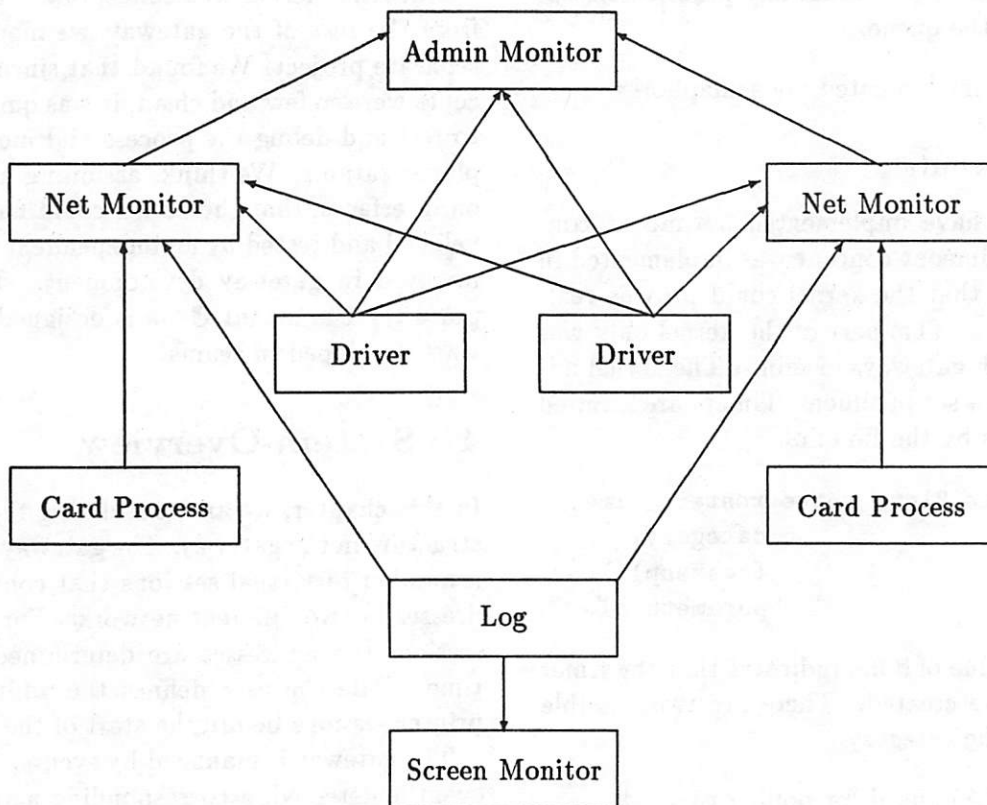


Figure 1: Process Structure

- IN\_CONNECTION
- IN\_EXPEDITED\_DATA
- IN\_NORMAL\_DATA
- OUT\_DISCONNECT
- OUT\_CONNECTION
- TIMEOUT\_EVENT

Only the events prefixed by "IN" originate from the network hardware. The events prefixed by "OUT" are generated in order to synchronize the two drivers. When e.g. a connection is received by one of the card processes an IN\_CONNECTION is generated in the corresponding driver. In order to create a virtual session between the two networks, the driver attempts to call a function connect in the other network monitor. If the

connection succeed an OUT\_CONNECTION is generated in an other network monitor to start the datatransmission. The analogue situation takes place when a session is about to be closed down. IN\_NORM\_DATA and IN\_EXP\_DATA are generated when a normal data packet or an expedited datapacket is received. A TIMEOUT\_EVENT is created in the connect-receiving process, if the session is inactive for a user defined amount of time. When a disconnect is detected by a card process a special event HANGUP is generated in the driver for the other network. If the driver is flow controlled, the HANGUP event will cause an exit from any sending operation, so that a DISCONNECT event can be received.

The events are managed in the network monitor superclass. Subclasses of the network monitor take responsibility for the network activity,



see section 5.3.

## 4.2 Driver Process

We will now describe the driver processes in further detail. When the driver is inactive it is waiting in a member function of the network monitor, `wait_event`. When an event occurs, the driver returns the event from the `wait_event` routine, and the corresponding action takes place. The driver process is also responsible for presenting a menu for the user in order to enable the user on a terminal session to select the target address. The menu is implemented as a separate class.

The driver processes are interacting with the `admin` class in order to receive pointers to the network monitors, and the `menu` class. The `admin` class is also synchronizing the opening and the closing of the session. The driver process has the following pseudo-code:

```
driver::main()
{
    forever {
        get net monitors and menu-classes.
        do
        {
            wait on connection event
            connection = try to connect;
            if ( connection )
                start datatransmission;
        } while ( connection );
    }
}
```

Every driver is listening for connect requests on each of the networks. When a driver gets at `IN_CONNECTION` event it tries to connect to the other network, or if the session is allowing connection only in one direction, the connect is rejected. When the other driver receives an `OUT_CONNECTION`, a dummy connect enables the start of data transmission, in both directions.

Every time a session is started, the drivers get their network monitors from the `admin` class, in order to allow easy reconfiguration.

Note that the network driver is written totally independent of the physical network. The process is interacting through network monitors which have an abstract interface, and the two drivers are exactly the same. The real network activity first take place in the virtual procedures in the subclasses of the network monitor descriptors. The variables for both driver processes are declared as private variables in the driver process, and in this a driver only obtains access to its own variables.

## 4.3 Card Process

The detection of the events is managed by the card process. The card process supervises the physical hardware, and creates an event in the appropriate network monitor, when the event is detected. Special action takes place when a disconnection is detected. In order to close down a flow controlled session, a special event `HANGUP` is generated in the other session, in addition to the normal `IN_DISCONNECTION` event. When a `HANGUP` is generated in a network monitor any send operation is immediately aborted.

The card processes for a physical network are a subclass of the following class:

```
card_process::report( net, physid, event )
{
    report event to net monitor
    corresponding to ( net, physid );
    if ( event == IN_DISCONNECTION )
        report a HANGUP to the
        other net monitor;
}
```

The card process is a subclass of the `process` class as mentioned in section 5.3. Every subclass of the `card_process` has to define a virtual main procedure, which provide the main algorithm of the process. We have in this way used C++ class hierarchy to hide network dependent parts of the process in the subclasses of the card process descriptor.

## 4.4 Monitoring Process

A monitoring screen is attached to the gateway. It is here possible for the user to see some statistics about the current state in the gateway. The monitoring process is responsible for showing the information and managing the screen and keyboard. The monitoring process accesses entries in the network monitors to obtain the current state of the different sessions. It is e.g. possible to see the following information.

- Number of bytes received.
- Number of bytes send.
- Time for last read.
- Time for last write.
- Time for last connect etc.

This process is independent of the physical network. The log process gets its input from the network monitor superclass, see section 5.3. The statistic variables are updated in the network monitor superclass when the virtual function in the subclasses is called.

## 5 Data Structures

In this section, the most important data structures and their use in the gateway will be described.

### 5.1 The State Machine

The basic protocol in the gateway is based on a finite state machine. Some of the generated events force a state change in the state machine. The state machine is outlined in figure 2.

### 5.2 Administrator

All information essential for the gateway processes is stored in a central class called `admin`.

The `admin` class is used in the following cases

- Assignment of device monitors to drivers

- Assignment of a dialog class to drivers <sup>1</sup>
- Mapping between sessions

The assignment of device monitors to drivers is made by the following function call

```
get_request(session_id,  
            &{2 session monitors})
```

where `session_id` is the identification of the session. The identification is used to administrate connects / disconnects / timeouts.

The purpose of `get_request` is to make it possible to reconfigure the gateway while it is running. If the gateway is reconfigured two things happen

1. A special RECONFIGURE event is generated.
2. When a session is closed down, it will be updated by a `get_request`.

This two-level protocol is necessary because it should be possible to have active sessions during a reconfiguration. When `get_request` is called, the session will be locked. This is necessary in order to protect against unintended HANG\_UP-events in complement.

The following must be used when we want to start a session or terminate one:

```
boolean open (session_id)  
boolean close(session_id)
```

The functions ensure a synchronized start and stop of sessions. `close` is used when a driver wants to terminate a session, this might be due to a TIMEOUT event. `close` returns a value indicating that the driver may initiate a disconnect or that it may not. `open` is used when a driver wishes to start a session. `open` ensures that two drivers do not send an OUT\_CONNECTION to each other at the same time. This might occur if both drivers

<sup>1</sup>When a terminal connects to the gateway, a dialogue is started in order to determine what the user wants to connect to.

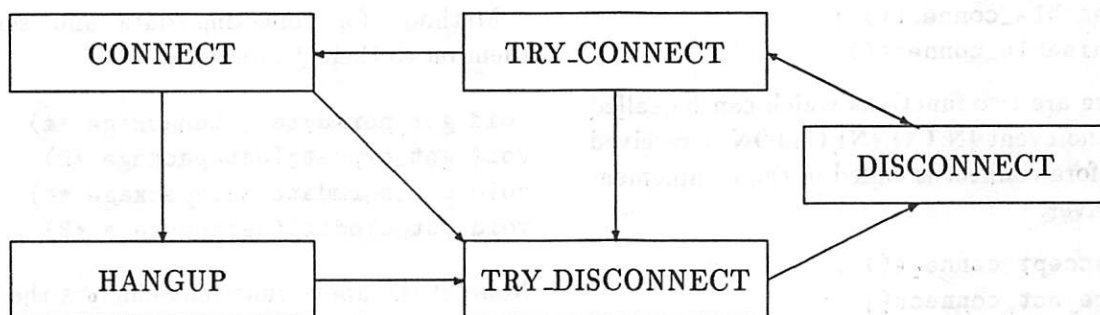


Figure 2: Finite State Machine

were to receive a connect at the same time. `open` and `close` are called from the driver processes.

When a user connects to the gateway from a terminal, the gateway must act as a switch-board. In order to help the user deciding where to connect, a dialogue is initiated. A call for getting the correct type of dialogue is made, `get_dialog(&dialog)`. `dialog` is a reference to a dialog class, which can be used when a user dialogue is needed.

If a card process is to communicate events with the driver of the other network, it must have a pointer to the session for the other network. The following function implements the above-mentioned:

```
boolean complement(net_type, phys_id,
                  &*session)
```

the return value implies whether it was possible to find a complementary session. `session` is the address of the other session which may be called when there is an event we want to communicate.

### 5.3 Network monitors

The concept of a network monitor is derived from the concept of device monitors. Device

monitors are described in [BH77]

A network monitor is a subclass of the session class. Session contains functions for connection, sending and receiving data. When these functions are called in the superclass, the corresponding hardware specific virtual function is called in the network monitor class. The corresponding function is given the same name as the superclass function prefixed with an `_`. For example `connect` will be named `_connect`. The state machine described in 3.1 is taken care of in the session class functions.

The entries in the session class are as follows.

```
boolean connect(char *address)
```

Make a connect for the calling session, using `address` as the connect address. The network monitor enters the `TRY_CONNECT` state while the connection is being established. If the connect attempt is successful, the state will be `CONNECT`. If a connect is made, an event called `OUT_CONNECTION` is generated. `OUT_CONNECTION` indicates to the complement driver that a connection was established. Connect may only be called when the driver has obtained permission to do so from the administrator.

The following two functions are used to allow or disallow connect from remote network units:

```
void enable_connect() ;
void disable_connect() ;
```

There are two functions which can be called when the event IN\_CONNECTION is received and before connect is called in the complementary driver.

```
void accept_connect() ;
void reject_connect() ;
```

The last function is used for synchronizing when we close down a failed connection attempt. If the last function is called, the network monitor enters the DISCONNECT state.

The following function is supplied for disconnection purposes void disconnect() ; This function may be called when the admin has allowed a shut down of the session. During a shut down the network monitor enters a TRY\_DISCONNECT state. When the shut down operation is finished the state will be changed to DISCONNECT.

When the event IN\_DISCONNECTION is received, the function void accept\_disconnect() is called. The function call will change the network monitor state to DISCONNECT.

The interface to the event system is

```
void report(event_type)
event_type wait_event()
```

The report function is called when another process wants to generate an event. The wait\_event function puts the calling process to sleep in a queue until an event arrives. Normally when the report procedure is called, a state change will not take place, unless the argument is one of the following values

- TIMEOUT changes the state to HANGUP
- HANGUP\_EVENT changes the state to HANGUP
- IN\_DISCONNECTION changes the state to TRY\_DISCONNECT
- IN\_CONNECTION changes the state to TRY\_CONNECT

Methods for collecting data and sending them on to their destination.

```
void get_normdata(datapackage *)&
void get_expdata(datapackage *)&
void put_normdata(datapackage *)&
void put_expdata(datapackage *)&
```

None of the above functions changes the state of the network monitor.

The network monitors are, as mentioned before, implemented as subclasses of session. There is a corresponding operation in the subclass. for each of the operations in the base class. The name of the function is just prefixed by an -. Apart from the state machine, session also handles some common variables. Each following of the events

- IN\_GOING\_COMPLETION
- OUT\_GOING\_COMPLETION

has associated a queue. When a subclass is in a state where it wishes to wait for any of the above events it just make a call like the following

```
queues[IN_GOING_COMPLETION]->Delay()
```

The superclass is responsible for waking up the process when the event happens.

The network monitors must also keep track of timeouts. The timeouts are necessary to prevent session being inactive for long periods of time, e.g. users forgetting to log out. The session class starts timeouts when an incoming connect is received and accepted. This is implemented via timers implemented in the kernel. Each timer has associated a function which is called at the moment of timeout. The function usually awakes processes sleeping in e.g. a send operation. At the network monitor level timeouts with very small timeout values are handled. In other words, polling takes place here via VIRTUAL timers. These timers are checked during each preemptive context switch and every time there is nothing else to do.



## 5.4 Did C++ help ?

The main advantage using C++ was that we were able to factor out the netspecific details in network monitors. This helped us a great deal while trying to understand what was going on during a session without bothering about hardware details. We were also able to implement the necessary hardware-dependent subclasses independently of each other. The implementation is in fact so general that we could write an interactive test driver and test our implementation against that. Thus, we were able to test and verify the two hardware monitors independently of each other. Integration was now only a matter of including different header files.

## 6 Experiences

In this section, we will describe our experiences using C++.

### 6.1 Problems Using DOS

On a PC where memory is a very limited resource the translation of C++ programs poses some serious problems. When there are many include files, the translation fails due to insufficient stack space. The main program had to include most of the class interfaces in order to initiate the classes properly. We had to use constructs like

```
extern admin_initialize() ;
....
admin_initialize() ;
```

in the main program. In the implementation part of the administrator, we declared a C function `admin_initialize` with the following body

```
void admin_initialize()
{
    admin.init() ;
}
```

where `admin` is an administrator variable.

This is not something to be proud of. Future ports of C++ to DOS must address the memory problem one way or another. Otherwise it will be very difficult to write nontrivial C++ programs under DOS.

### 6.2 We Would Like

The only thing we really missed during development was a top down calling convention in the class hierarchy. The method for this could look like the `inner` construct in Simula.

The idea is that given

```
class base {

    f() { some code
        inner ;
        more code
    }
}
```

where `inner` calls a function named `f()` in derived

```
class derived : base {

    f() { code specific for derived }
}
```

Given that feature the naming convention simulating `inner` in section 5.3 is not necessary any more. The code would thus be easier to read.

### 6.3 Network Independency

One of our primary objectives with this project was to write something so general that changing of network hardware would only mean changing one or two C++ modules. With the concept of net monitors we think that we have achieved the objective. At the time of writing this article we are starting a new gateway project. Our changes in the existing code should be limited to a few modules.

We also find that the system can be easily extended. One could consider building a gateway interfacing more than two networks. In

our model, this can easily be managed by the admin class, by returning net monitors to the driver, which can implement different types of networks. With the driver concept we have obtained a network independency through the use of abstract interfaces to the net monitor, and thus we will not get any problems by choosing from a set of different types of net monitors, when we return a pair of net monitors to the drivers.

It is also possible to make a system which allows dynamic reconfiguration at runtime. In fact, we have introduced an event RECONF, as described in an earlier section which prepares the system for dynamic reconfiguration.

Of course our system might have been implemented by programming in C, too, and we might have achieved the same level of net independency. But at lower levels we would have problems simulating the classes and the virtual functions from C++.

## 7 Conclusion

The most important conclusion we were able to draw from this project was that the encapsulation and modularity concepts provided by C++ make it eminently suited for parallel development and debugging.

It was also very easy to translate our system model into C++. This translation from model to code would have taken more time if we had used a more traditional language like C. We observed that program development using C++ meant more analysis before coding and less coding afterwards.

The compact form of C++ programs also meant fewer lines of source code which imply fewer errors.

All in all we are very satisfied with C++ as a systems programming language.

## Acknowledgements

The authors wish to thank Lisa Wagner, Jutland Telephone, for her proof reading of our

manuscript.

We also wish to thank Jens Palsberg Jørgensen, Computer Science Department University of Aarhus, and Birger Nielsen, Jutland Telephone, for their reading and commenting on drafts of this paper.

## References

- [BH77] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall, 1977.
- [BLL88] Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Pestro: A system for object-oriented parallel programming. *Software-Practice and Experience*, August 1988.
- [Hoa78] C.A.R. Hoare. Monitors: An operating system structuring concept. In David Gries, editor, *Programming Methodology*. Springer-Verlag, 1978.
- [Hol87] Allen Holub. A preemptive multitasking kernel and more mean subroutines. *Dr. Dobbs' Journal*, December 1987.

# An Exception Handling Implementation for C++

Michael D. Tiemann  
March 3, 1989  
(GNU C++ version 1.37.1)

## Abstract

This paper outlines a design for an exception handling mechanism that has been implemented in GNU C++, a free, native-code C++ compiler. Various alternatives for handling exceptions are presented. An abstraction is derived which solves problems with existing C/C++ solutions. The abstraction is flexible, type-safe, works in a mixed language execution environment. Two implementations are presented: one which can be implemented by a C++ to C translator, the other which can be implemented to run efficiently under a native-code compiler. This paper concludes with a brief survey of other exception handling designs.

## 1. Introduction

Program behavior is divided into two categories: normal operation and exceptional circumstances. The C++ programming language was designed to support programming under normal conditions. Unfortunately, exceptional circumstances (hereafter called *exceptions*) do arise, and when they do, most C++ programs are poorly equipped to deal with them, mainly because doing so is hard without language support.

This paper presents an extension to the language which supports programming in the presence of exceptions. It provides the programmer with a disciplined way of dealing with exceptions that is portable, type-safe, works in a mixed-language execution environment, and can be implemented to execute efficiently.

The presentation begins with a motivation for adding exception handling to C++. With that background, it then shows how the design of a particular exception handling design evolved. The use and utility of this design is then shown. Finally, the design is targeted to two different compiler

platforms: a C++ to C translator and a native-code compiler. The paper concludes with a brief comparison of this exception handling design with previous designs.

## 2. Why Handle Exceptions?

Reliable programs must be able to expect the unexpected. When a subroutine encounters a situation with which it cannot cope, three alternatives are traditionally used in C/C++:

1. Set a flag and return an error code;
2. Abort the program;
3. Print an error message and continue, hoping for the best.

These alternatives are not attractive for these respective reasons:

1. The domain of every return type must include an error code, and every value returned by a function must be checked against this code;
2. The program, at some higher level, may be prepared to deal with the problem;
3. A program that has been poisoned is dangerous; it should be cured or killed.

We therefore conclude that existing C/C++ mechanisms are not sufficient to deal effectively with exceptional events (such as running out of memory or trying to write to a file that no longer exists). Given the propensity for exceptional events to occur, and a desire to write programs that don't crash (such as operating systems) a more reliable mechanism is needed.

## 3. An Abstraction for Exception Handling

The characteristic of the exceptions treated here is that they show up in one place, but must be handled in another. While it is perfectly valid to consider an exception within a C function that must be handled in another place within that same function, we consider for now that such a case is largely solved by using a `goto` statement. What has not been solved is when a function `f` is active, and wants to raise exception `EX` to be handled by one of the functions that (transitively, at the time `f` is active) called `f`. I.e., we are concerned with exceptions that cross function-call boundaries. In this paper, solutions for inter-function exception handling can be trivially generalized to intra-function exception handling.



We begin by examining the properties of function calls and control transfer.

### 3.1 Solutions in C

The C programming language, and by extension C++, has a very simple function-call model. Informally, a C function call takes a (possibly empty) list of arguments, and returns a (possible void) value. The point of return is always to the point of invocation.<sup>1</sup> In the parsimonious world of C and C++ programming, error values or exception values are not usually in the domain of the return type, so it is often not possible to “return” an error code. To handle exceptions, then, the return mechanism must be augmented.

Multiple return values (i.e., a normal return value and an error code) can be simulated by returning aggregate values, though with some performance penalty. The use of aggregate return values would also clutter code with temporary variables: what was once the use of a function call as an expression would have to become the initialization of a temporary variable via a function call, followed by a test of that variable’s return status code. If the return status code signified that things are ok, the remainder of the temporary would be used for the function return value. If the return status code signified an exception, then exception handling would have to be initiated at that point.

Handling exceptions by using multiple return values would change code that looks like this:

```
int f (int), g (int), gronk (int, int);  
int i = gronk (f (1), g (2));
```

into this:

```
enum error_code { OK = 0, ... };  
struct aggr { error_code err; int val;};  
void handle (aggr);  
aggr f (int), g (int), gronk (int, int);
```

---

<sup>1</sup> Except for `longjmp`, which is not really a C function anyway.

```

aggr t1 = f (1);
if (t1.err != OK) handle (t1);

aggr t2 = g (2);
if (t2.err != OK) handle (t2);

aggr t3 = gronk (t1.val, t2.val);
if (t3.err != OK) handle (t3);

int i = t3.val;

```

The overhead includes one extra value assignment and one extra conditional test per function call. Also, on many machines (depending mostly on the compiler), it might mean that all function calls would have to return values through memory rather than through registers. All in all, such a solution would add noticeably to execution time, and program size.

The use of side-effects is an alternative to returning aggregate values, as shown in the following code typifying the UNIX solution:

```

enum error_code { OK = 0, ... };
error_code errno;
int f (int), g (int), gronk (int, int);

int i = gronk (f (1), g (2));

// which function set ERRNO???
if (errno == EIEIO)
    moo ();

```

Notice that this technique is not reentrant, and highly error-prone, since the type system does not give the user a way to automatically check that s/he has tested all variables that a program might be using to store exception information. So while this solution has reduced apparent overhead, it has done so at the cost of code safety.

## 3.2 A Better Solution

The C function-call model is based upon a fixed, typed, immutable *continuation*. The continuation is the pair <return value, return address> and cannot be changed or extended by the programmer. It is maintained by the compiler, and incurs very little overhead: on the average machine the cost is a push to save the return value, and the setting of a register to communicate the return value. The continuation is activated by a simple "return from subroutine" instruction. All this simplicity is nice for the compiler writer, but a pain for the programmer: in order to control

behavior of the program across function call boundaries, the programmer must set state variables and interpret them manually (i.e., compare a global variable against zero, etc.).

With a more general continuation model, the programmer would have the ability to pass control structures across function-call boundaries. In particular, it would be possible to have a function return to multiple places by building a continuation which encapsulates different points of return:

```
// cont is a first-class continuation
int f (int, cont), g (int, cont), gronk (int, int, cont);
cont k1, k2;

int i = gronk (f (1, k1), g (2, k1), k2); ... // normal return
// ok to use 'i' here

k1 (int x):
// deal with one kind of problem involving X

k2 (int x, int y):
// deal with another kind of problem involving X and Y
```

where a function's C continuation (returning to the point of invocation) is considered implicit.

This gives us the following advantages:

- continuations make control transfer for exceptions explicit;
- use of continuations can be type-checked;
- continuations provide an encapsulation for environments which must share information;
- the same continuation can be used in many places within the program.

with the following disadvantages:

- additional overhead for function calls;
- nontrivial extension to C.

A good design would therefore provide the advantages without imposing the disadvantages.

## 4. Mapping Continuations to C

First-class continuations as provided by languages like Scheme are not easily added to C. However, one can approximate a first-class continuation using the `setjmp/longjmp` mechanism. The `setjmp` function establishes a continuation, and a call to `longjmp` “returns” to it. There are significant constraints as to how these continuations can be set up, and modification of these continuations is not defined. Using C, we have:

```
int f (int, jmp_buf), g (int, jmp_buf), gronk (int, int, jmp_buf);
jmp_buf j1, j2;

if (setjmp (j1)) goto k1;
if (setjmp (j2)) goto k2;

int i = gronk (f (1, j1), g (2, j1), j2);
...
// normal return

k1: // deal with one kind of problem

k2: // deal with another kind of problem
```

As can be seen, the syntax is clumsy, requiring lots of “continuation” declarations in addition to the calls to `setjmp` everywhere. Control transfer has been made explicit, but at the cost of code legibility. Because the programmer must still manually set up the continuations her/himself, it is still all too possible to write code that will go haywire when calling `longjmp` on an uninitialized `jmp_buf`. It would also be easy to call `setjmp` on the wrong `jmp_buf`, clobbering a valid continuation.

Using this model, there is no easy general solution to passing information from the exception raiser to the exception handler without extra mechanism. Non-reentrant techniques are rejected because they defeat the requirement of this design. Non-portable solutions, such as using the return value of `setjmp` to pass back a pointer to a block of arguments, cannot be accepted either.

Finally, in passing around the continuation, there is still the overhead of passing an extra parameter to every exception-handling function call.



## 5. The Need for Special Syntax

To implement an exception handling mechanism which is compatible with C++, two criteria must be met: exceptions must interact properly with existing C++ constructs and they must be implementable in an efficient way. The major interaction which must be preserved is that when a binding contour is exited, destructors for objects in that contour must be called. As it is, `longjmp` makes no such guarantees, so `longjmp` alone is not sufficient.

Due to a concession made by the ANSI C committee, `longjmp` may return to an environment established by `setjmp` in which registers at the point of the call to `setjmp` will have different values than just after the `longjmp`. For this reason, `setjmp` and `longjmp` can interfere with, if not completely defeat register-based optimizations that the compiler might perform. Therefore, it is desirable to be able to base an exception handling implementation using continuations that can be implemented by something other than `setjmp` and `longjmp` (though `setjmp` and `longjmp` will certainly work).

A system which has a high-level notion of exception handling is now considered. For this system, `setjmp` and `longjmp` are used only to describe semantics; there is no stipulation that they must be used.

In the following code, the `try` block establishes an exception handler. Exceptions raised within the `try` block will be handled by handlers appearing at the end of the `try` block. In this model, EX and EY name exceptions, and the `catch` statements combine with them to denote continuations. That is, if a function raises the EX exception, then control will logically "throw" to the handler labeled `catch EX`. Similarly for EY.

```
int f (int) raises EX, g (int) raises EX;
int gronk (int, int) raises EY;

try
{
    int i = gronk (f (1), g (2));
    ... // normal execution
}
catch EX // deal with one kind of problem
{
}
catch EY // deal with another kind of problem
{
}
...
// normal return
```

Implementing the example code using the above model makes control transfer explicit, satisfying the first of the three desiderata.

The model is strongly typed in that the declaration of a continuation is matched with its handler, so the programmer is not in a position to make mistakes with `setjmp/label` pairs. Additionally, functions which raise exceptions can make that fact explicit in their type signature, aiding program documentation and making it possible for the compiler check consistency of interfaces.

It should be noted that by a simple translation process, this structure can be implemented with C semantics using `setjmp` and `longjmp`<sup>1</sup>, hence no extensions to a C-based model are necessary. Furthermore, because the model is used specify exception-related semantics without specifying a corresponding implementation, optimizations are possible when considering target implementations beyond the C language. In particular, a mechanism can be implemented in such a way that there is no overhead for establishing `try` blocks.

## 6. Extensions to the Model

The model of the previous section lacks a means to pass information from the exception's raiser to the exception's handler (beyond the raised exception's name). The model also does not treat relationships between exceptions, specifically subclassing. The following section explains the key observation which unifies the solution we have so far with a C++ implementation.

### 6.1 The Environment Abstraction

A binding contour maps names to locations, and is the mechanism by which global variables, parameters, and local variables are supported in C and C++. Characteristics of these binding contours are:

- they *nest* in a particular order;
- they are *static* (as opposed to *dynamic*);

---

<sup>1</sup> In fact, such an implementation exists in GNU C++ 1.37.1

- they are not first-class objects.

These characteristics imply that binding contours cannot be passed from one function to another, and that they cannot be modified at run-time. A function *f* can pass information to another function *g* only by modifying the values mapped by *f*'s binding contour, but it cannot pass its own binding contour for *f* to use. For completeness, it is noted that parameter lists can be passed around via the *varargs* mechanism, but that that mechanism is very primitive, and does not permit parameter locations to be named outside their containing function's scope.

In the Scheme world, environments and continuations go hand in hand. An *environment* is a generalization of a binding contour. When a continuation is called, the arguments to the continuation become part of the resulting (callee's) environment. A similar feat can be performed using objects.

## 6.2 Sharing Environments

It is appropriate to begin this section with the age-old question "what's in a name?". Specifically, is the name of an exception sufficient to pass to the handler all the information needed to handle the exception? In general, the answer is no. It will certainly do for coarse-grained exception handling to have the name encode all that needs to be known, for example *DIV\_BY\_0*. But consider the case of a process requesting exclusive access to a set of resources. If some subset of those resources are unavailable (since granting them might cause a deadlock), the requestor might like to know which ones are causing trouble, rather than handling an otherwise uninformative *EDEADLOCK* exception.

To review C-style mechanisms which will not solve this problem, we begin by noting that we have already rejected the simplest mechanism of encoding semantic information in the exception name.

The use of global variables to hold the exception environment is rejected because it provides poor encapsulation for the exception handler (thereby violating the object-oriented programming paradigm). It is also not reentrant, making it difficult if not impossible to use in a parallel processing environment.

Passing a local environment down the call chain to be filled in by the raiser of an exception would add overhead to all calls to functions active between the raiser and handler. The overhead must be paid whether or not exceptions are raised at run-time. Such a solution reintroduces the efficiency problem that we worked so hard to remove. It also requires the programmer to write and maintain a lot of extra code. The extra work means that the programmer will probably not

bother with it, and choose instead to write programs limited to name-based exception handling. If the difficulty of using the exception handling mechanism discourages programmers from using it to solve common exception-based problems, then its design should be rethought or scrapped.

## 6.3 Environments and Objects

In general, lexical closures, hence environments, can be first-class objects, and can therefore be shared by functions in the same way that a pointer can be shared. This section describes how the environments between exception raiser and exception handler can be shared using objects.

Objects are interesting in that they define a lexical closure which is distinct from the standard C lexical closures (the global binding level, the function parameter level, and inner program blocks). In contrast to these closures, which are static, the lexical closure of an object is dynamic. In other words, C binding contours stay in one place; to access their elements, the user must be brought into their context. On the other hand, the environment of an object moves around with the object, and to access its elements, the object need only be brought to the user. For this reason, information encapsulated within the environment of an object can be shared between two viewpoints separated across function boundaries, while information encapsulated in static environments (except the global one) cannot.

It is precisely this use of encapsulation that makes objects a uniquely suitable vehicle for solving the communication problem between exception raiser and exception handler. No other part of the language solves this problem so elegantly:

```
exception { int i; char *p; } EX;
exception { String s; } EY;
int f (int) raises EX, g (int) raises EX;
int gronk (int, int) raises EY;
```

```
try
{
    int i = gronk (f (1), g (2));
    ... // normal execution
}
except ep
{
    EX {
        // use exception parameters
        if (ep.i) printf (ep.p);
    }
    EY {
        // use other exception parameters
    }
}
```



```

    raise EY (ep.s + " raise ");
}
default {
    // default exception, parameters are invisible
    raise ep;
}
...
// normal return

```

We can therefore pass information from exception raiser to exception handler by passing an object encapsulating the raiser's environment as part of the continuation.

## 6.4 Exceptions and Inheritance

Inheritance has been identified as one of the central concepts of object oriented programming. In this exception handling design, types play a key role in the specifications and use of exceptions. This design would be incomplete without discussing how these new kinds of types interact with or relate to inheritance.

Just as with normal programming, one can imagine uses for exceptions which can be handled by general handlers (base handlers) or can be handled by more specialized handlers (derived handlers). We first consider how normal inheritance is used in normal programming contexts. We then show how inheritance is used in under exceptional circumstances, and conclude with a design that satisfies our minimal requirements.

As pointed out previously, normal type relations are static. Objects of base type and objects of derived type can be constructed such that the type of the object (often implemented by its virtual function table) remains constant throughout the object's lifetime. In particular, how the object reacts to method calls is invariant of the object (except during construction and destruction, which are special cases).<sup>2</sup>

For exceptions, the way in which exceptions are handled is not an invariant of the exception; rather it is a function of both the exception being raised and the dynamic context (i.e., the current lexical contours and/or current call chain). Thus, it is not possible to talk of inheritance strictly as a function of the exception type.

---

<sup>2</sup> Actually, this is not true for non-virtual methods, but we consider such cases anomalies.

The reasons for using inheritance in handling exceptions are the same reasons for using inheritance in normal programming. Namely, to support (and promote) code reuse by writing general handlers which can be specialized as programs evolve. For example, when using an I/O package, one should expect to handle I/O errors. Initially, the program will handle very general I/O errors, but as the program evolves, it might handle them at a finer grain of detail. If the package raises `FileScrambled` error, it would be nice to be able to catch either `IO_error`, `File_error`, or `FileScrambled_error`. With inheritance of exceptions, one handler could catch all three, or the most specialized handler could be applied. Thus it is easy to write a prototype program which is still robust.

Seeing the utility of inheritance of exceptions, the question arises of how to support it. Two possibilities spring to mind: the first is to implement inheritance of exceptions directly; the second is to use inheritance as provided by objects. We consider both implementations and choose the one which delivers all the features we want without compromising the efficiency of the implementation.

The interesting implementation of inheritance for objects is provided via the objects virtual function table(s). Irrespective of where a virtual method call is made to an object, the object responds to the call in the same way. Thus the virtual function table serves to encode part of the object's behavior. (In contrast, non-virtual method calls are more a function of the object's lexical environment—whether the object is seen by the compiler as a being of base or derived type.)

Since the handling of exceptions is a function of the dynamic call chain, an exception cannot itself encode an action to perform in the event that it is raised. An exception therefore needs a means of identifying itself so that the run-time exception mechanism can know which handler should handle the exception. Without inheritance of exceptions, the name of an exception was sufficient to identify the exception. With inheritance of exceptions, a more complicated encoding is needed.

An inheritance tree can be encoded as a tree of names in the inheritance tree. This tree, like the name, can be built at compile or link time, and can even take advantage of structure-sharing techniques. Multiple inheritance can also be handled by constructing a directed acyclic graph (DAG) instead of a tree. The search mechanism can optionally be enhanced to optimize a search of the DAG; otherwise, it will search it as a tree, visiting some subgraphs more than once.

As usual, the search mechanism must be written to find the best match among a list of potential handlers. When the exact handler appears, that match must be found first. After that, it is up to the exception matcher to find the best alternative. In some cases, an exact match against a derived handler might be preferable to a match against a base handler closer on the call chain. In other cases, the first handler that can match at all is the wanted one. Such flexibility can be achieved by

associating a matcher with the exception being raised.

This implementation provides the features of inheritance that we want, but at a price which is the complexity of the search mechanism. Since exceptions are considered exceptional, the speed with which an exception can be handled is not of primary importance. But it is not desirable to add a lot of code and data overhead to implement features that will not (or need not) be used.

A second implementation is much simpler. Under this implementation, inheritance of exceptions is provided via existing mechanisms. Exception types *per se* are not inheritable, but the parameters to the exceptions can be types which obey inheritance. Thus, the system works with a small number of primitive, general exceptions, and special behavior is provided by objects which may be parameters to exceptions.

In C++:

```
exception { class base *b; ... } IO;
class File : public virtual base { ... };
class Network : public virtual base { ... };
class NetworkFile : public File, public Network { ... };

int foo ()
{
    ...
    try
    {
        fd = open ("bar", "xyzy");
    }
    except ep
    {
        IO {
            (ep.b)->handle (fd);
        }
    }
}
```

The second implementation provides access to the use of inheritance for exceptions without requiring any further extension or complication of the original model. It is also a proper subset of the implementation which permits explicit inheritance of exceptions. Since it provides the features we want without extra complexity, and since it is upwards compatible with a more complicated mechanism, we choose it as the best way to provide inheritance to users of exceptions.

## 7. Implementation

This section described two possible implementations: one for a C++ to C translator and one for a native-code C++ compiler. The exception mechanism presented in this paper can be implemented as follows:

The declaration of an exception such as:

```
exception { T1 p1; T2 p2; } EX;
```

results in the following:

```
long exception_EX; // used to identify exception.

class EX {
public:
    T1 i1;
    T2 i2;
    EX (T1 p1, T2 p2) { i1 = p1; i2 = p2; }
    ~EX () {}
};
```

The representation of an exception handler is as follows:

```
class ExceptionHandler {
    // enclosing handler
    ExceptionHandler *prev;

    // see setjmp/longjmp
    jmp_buf handler;

    // exception name
    long *name;

    // pointer to parameter object
    void *parameters;

    // constructor and destructor for handler
    ExceptionHandler ();
    ~ExceptionHandler ();
};
```

The constructor for an exception handler initialized the fields and links the new handler on the top



of the exception handler stack. The destructor unlinks it and releases the storage for the handler object.

The try statement expands into the following code:

```
{ // start of new block for handler
  ExceptionHandler eh (); // allocation handler
  if (setjmp (eh.handler) == 0)
  {
    // statements
  }
  else {
    exceptionHandlerStack = exceptionHandlerStack->prev;
    if (eh.name == &exception_EX) {
      // handler for exception EX
      // access parameters by casting eh.parameters to class EX*
    }
    else if /* look for other exceptions ... */
    else {
      // either default handler or reraise using
      // eh.name and eh.parameters.
    }
  }
}
```

The raise statement expands into the following code:

```
exceptionHandlerStack->parameters = (void *)new EX (parameters);
exceptionHandlerStack->name = &exception_EX;
DoDestructors (); // run all destructors necessary
longjmp (exceptionHandlerStack->handler);
```

For a native code compiler, a similar structure is used, but calls to `setjmp` are not needed. Instead, the compiler notices try statements, and emits code indicating their boundaries. The linker builds tables which map program counter ranges to specific handler addresses. When an exception is raised, the run-time system looks in the tables to find the appropriate handler, and sends the execution thread to that location. Thus, the overhead of a try statement is reduced to nothing at runtime. The overhead of a raise statement might be higher, but since raise is expected very infrequently, this is not considered a problem.

## 8. A Survey Other Designs

The differences between exception handling implementations can be distinguished in two interesting ways: differences in policy and differences in mechanism. This section is far too brief, but is included in the hopes that it provides some completeness.

In the Mesa system, exception handling was introduced for two primary purposes. The first was to make it easy for the programmer to be able to program as though things that were not really supposed to happen could be treated as though they never would, and secondly, as a form of documentation of behavior. For example, when the programmer wants to use the Mesa storage allocator, s/he can program as though it never returns NULL. The exception handling mechanism handles the rare instances when it does, but otherwise, there is not need to test the return value against NULL. Mesa takes advantage of this fact, and optimized the implementation so that overhead is actually reduced in the normal case (though raising exceptions might be very expensive). In terms of program documentation, explicit language constructs clearly differentiate normal from exceptional program conditions.

In languages like CLU, exceptions are more a control structure than an exception handling mechanism. They are implemented very efficiently, and extensive analysis is performed by the compiler and the linker to transform the exception control construct into a simpler one (like `if-then-else`). In CLU, a common use of exceptions is to exit from a loop iterating across an array. The loop never explicitly tests the iteration variable against the array bounds. When the Bounds exception is raised (by accessing an element beyond the legal bounds of the array), the loop is exited. In this respect, one cannot reliably tell that an exception is really all that exceptional.

In LISP systems, exceptions implemented with `CATCH` and `THROW` primitives, which are similar to the `catch` and `raise` primitives in this C++ design except that `THROW` does not take arguments whereas exceptions raised in this design can. One interesting feature of the LISP exception handling mechanism is the interaction between signalling an exception and the `unwind-protect` construct. The way in which LISP systems must "clean up" their frames before exiting them is also a requirement in C++ where destructors must be called for objects that were constructed in that scope.

In Scheme, continuations are provided naturally by the language. Exception handling is therefore provided directly to the user. A global error continuation handles generic errors (such as trying to take the `CAR` of an atom). The programmer can handle error exceptions in special ways by establishing a new error continuation has as one of its continuations the previous error continuation. Thus, the exception handler stack is built from control constructs rather than data.

The other C++ exception handling proposal that has appeared to date (Koenig and Stroustrup) takes a very different view of exceptions. The major differences between that design and the one presented in this paper are with the way exceptions relate to the type system and the object system. Koenig and Stroustrup chose to make exceptions variables, and extend the language only to catch exceptions. They assume that `exception` is a base type for implementing exception types, and that the `raise` method of that type is sufficient to raise an exception. It proposes fewer extensions to the C++ language than this design, but in comparing the two, fewer is not always better. Because some of the mechanism can be directly affected by a programmers use of other language constructs, a program can interfere with the correct operation of the mechanism without the knowledge of the compiler. This leads to unsafe programs in precisely the area where we want to implement safety. They have indicated that a new design is currently in the works.

## 9. Conclusion and Acknowledgements

The C++ language must support exception handling to satisfy the needs of users who wish to write robust, reliable software that can still benefit from the advantages of object oriented programming. Several people have discussed implementations of exception handling which are compatible with C++ semantics. This paper presents more the evolution of an exception handling design

Michael Powell provided the initial exception handling model from which this paper was derived, with additional contributions coming from Jim Mitchell, Graham Hamilton, Jonathan Gibbons, Jim Kempf, and Steve Gadol. In addition to helping with the design, Graham and Jon slogged through early implementations, finding the strangest bugs...

Daniel Weise taught me about continuations, and gave me the insights I needed to understand their interactions with environments.

Doug Lea read early versions of this draft.

Bjarne Stroustrup and Andrew Koenig contributed to the discussion of these matters by providing me with a draft of their exception handling proposal. Jonathan Shopiro was also provided additional comments.

## References

American National Standard for Information Systems: *Programming Language C X3* Secretariat, 1990.

Kernighan, Brian and Ritchie, Dennis: *The C Programming Language*. Prentice-Hall, 1978.

Koenig, Andrew and Stroustrup, Bjarne: *Exception Handling for C++*. C++ At Work Conference Proceedings, 1989.

Lamping, John: *A unified system of parameterization for programming languages*, Phd Thesis, Computer Science Department, Stanford University, 1988.

Liskov, Barbara et al.: *CLU Reference Manual*, Technical Report 225, MIT Laboratory for Computer Science, 1979.

Mitchell, James G.: *Mesa Language Manual* CSL-78-1, Xerox PARC, 1978.

Pittman, Kent: *Exceptional Situations in Lisp*, Working Paper 268, MIT Artificial Intelligence Laboratory, 1985.

Steele, Guy L. and Sussman, Gerald J.: *LAMBDA, The Ultimate Imperative*. Memo 353, MIT Artificial Intelligence Laboratory, 1976.

Steele, Guy L.: *The Revised Report on SCHEME: A Dialect of LISP*, Memo 452, MIT Artificial Intelligence Laboratory, 1978.

Stroustrup, Bjarne: *The C++ Programming Language*. Addison-Wesley, 1986.

Tiemann, Michael: *GNU C++ User's Manual* Free Software Foundation, 1990.



# Runtime Access to Type Information in C++

John A. Interrante and Mark A. Linton  
Stanford University

## Abstract

The C++ language currently does not provide a mechanism for an object to determine its type at runtime. We propose the *Dossier* class as a standard interface for accessing type information from within a C++ program. We have implemented a tool called *mkdossier* that automatically generates type information in a form that can be compiled and linked with an application. In the prototype implementation, a class must have a virtual function to access an object's dossier given the object. We propose this access be provided implicitly by the language through a predefined member in all classes.

## 1 Introduction

Some applications need to know the names of classes, their inheritance structure, and other information at runtime. For example, the X Toolkit Intrinsics [3] define a customization mechanism based on class and instance names. With this mechanism, a user can pass a string to an application that is matched against instances of a named class. To applications, the string `"*Button*font:courier14"` means that the default font for all instances of `Button` is `"courier14"`. `InterViews` [2] is a C++ toolkit that supports the X Toolkit customization mechanism. Because the C++ language currently does not support access to any type information at runtime, programmers must write code in every `InterViews` class that defines the class's name.

Unfortunately, every class writer who needs runtime access to type information must invent their own conventions. These conventions make the exchange of user-defined data types between programmers difficult. For instance, both `OOPS` [1] and `ET++` [5] use macros in class definitions to provide a class's name and other information about class types. Neither library can reuse a class type from the other library without modification. Even if all libraries followed a standard set of conventions, these conventions still make writing classes more tedious.

Class writers must take two steps to eliminate these problems: (1) define a standard interface for accessing type information at runtime, and (2) define a way to generate the type information automatically. In this paper, we propose the class *Dossier* as the standard interface to type information, and we describe the implementation of a tool called *mkdossier* that generates a C++ source file containing dossiers. A programmer can then compile and link this file with an application.

A type declaration might not reflect the type of an object at runtime—the object could be a subclass of the declared class. Our prototype implementation therefore requires a class to have a virtual function which simply returns the dossier for its class. Adding a virtual function presents a problem because this change requires recompilation of source code defining or using the class. We propose that the C++ language be extended to provide a predefined (not reserved) member

---

Research supported by a gift from Digital Equipment Corporation and by a grant from the Charles Lee Powell Foundation.

---

```

class Dossier;

class DossierItr {
public:
    DossierItr();
    virtual ~DossierItr();

    boolean more();
    void next();

    // dereference through current element
    Dossier* operator ->();

    // coerce to current element
    operator Dossier*();
};

class Dossier {
public:
    Dossier(
        const char* name,
        const char* fileName,
        unsigned int lineNumber,
        Dossier** parents,
        Dossier** children
    );
    virtual ~Dossier();

    const char* name() const;
    const char* fileName() const;
    unsigned int lineNumber() const;
    DossierItr parents() const;
    DossierItr children() const;
    boolean isA(const Dossier*) const;

    // return array of dossier pointers for all classes
    static Dossier*const* classes() const;
};

```

---

Figure 1: Interface to dossiers

containing a pointer to a dossier. This extension would make type information available for existing classes without requiring changes to their source code.

## 2 Dossier interface

The name "Dossier" connotes detailed information about a subject. In this case, we want a dossier to be the repository for information about a type. Although a dossier could represent information about any type, in this paper we will only consider class types. Accessing information for non-class types requires compiler and language support, which we did not wish to undertake before defining an interface for classes.

Figure 1 shows the Dossier interface. The current interface provides limited information about a class: its name, the file and line number where the class is defined, and iterators to visit parent

---

```

void traverse(Dossier* d) {
    for (DossierItr i = d->parents(); i.more(); i.next()) {
        cout << "traversing " << i->name() << endl;
        traverse(i);
    }
    cout << "back to " << d->name() << endl;
}

```

---

Figure 2: Traversing ancestors using iterator

and children classes. The "isA" function determines whether a class type is a subclass of a given class type. The "classes" function allows the application to access all defined dossiers. Figure 2 shows a sample function that uses an iterator to print the names of all the ancestors of a class.

We anticipate extending the interface to include size information, names of members, and member functions. We concentrated on the minimal functionality so that we could investigate the mechanism without getting bogged down in too many details. When we settle on the final details, we can extend the functionality by replacing the standard "dossier.h" header file where Dossier is defined, updating the library implementation of Dossier, and providing a version of mkdossier that generates the additional information.

### 3 Dossier implementation

The Dossier interface describes what information is available for a class type; it does not say how the type information is generated. We expect a compiler or a special tool to automatically generate the representation of dossiers so that programmers need not manually define or update dossiers. Like the virtual function table used by most C++ compilers to implement virtual function calls, only one dossier representation should exist for each class in an application. We have implemented a tool called mkdossier that we use to generate dossiers just like we use makedepend, a tool developed at MIT, to generate Makefile dependencies.

Figure 3 shows the role of mkdossier in building an application. The build process calls mkdossier to scan all the source files and generate "\_dossier.h" and "\_dossier.c". Then the build process compiles the source files, including "\_dossier.c", and links them into an executable image.

Figure 4 shows a sample "\_dossier.h", the header file that declares the generated dossiers' names. The dossiers represent information about the sample classes "App," "ArgVec," and "CPP"; mkdossier defines each dossier's name by concatenating the prefix string "\_D\_" and the class type's name. The application can include this header file to import dossiers into application code by name as well as use the "Dossier::classes" function to import dossiers by address.

Figure 5 shows a sample "\_dossier.c", the source file that initializes dossiers with information about class types. Programmers can avoid having to manually define dossiers by compiling and linking this file into the application's executable image. At the end of the file, mkdossier generates a static array of pointers to all the dossiers defined in that file. The "Dossier::merge" function merges the static array into a global array containing pointers to all known dossiers so that the "Dossier::classes" function will work correctly even if the programmer compiles and links multiple "\_dossier.c" files into the executable image.

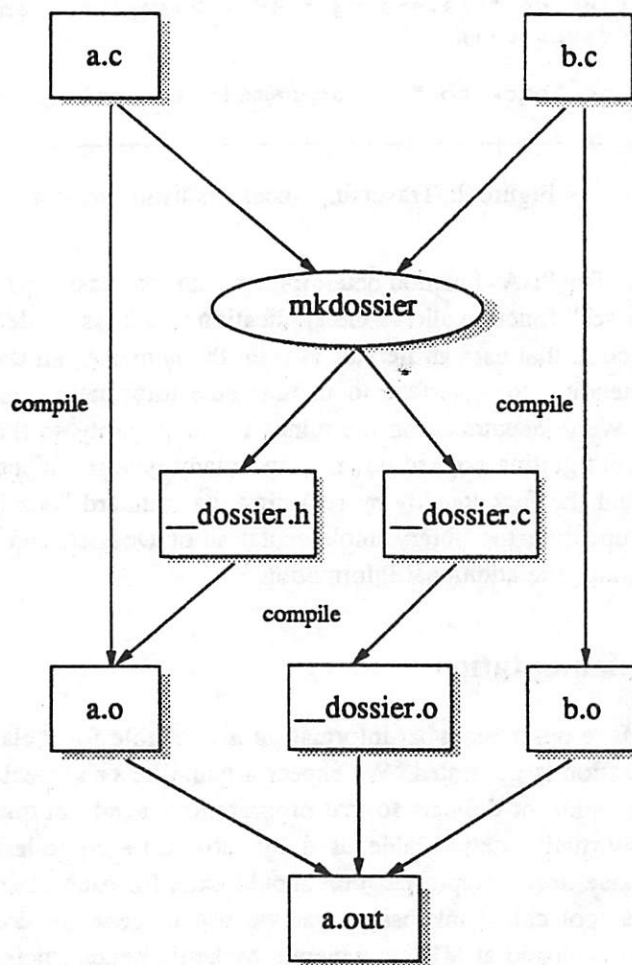


Figure 3: Generating dossiers with mkdossier

---

```

class Dossier;

extern Dossier  _D_App;
extern Dossier  _D_ArgVec;
extern Dossier  _D_CPP;
  
```

---

Figure 4: Declaration of dossiers



---

```

#include "__dossier.h"
#include <dossier.h>

static Dossier* __D_App_parents[] = { 0 };
static Dossier* __D_App_children[] = { &__D_CPP, 0 };

Dossier __D_App = Dossier(
    "App",
    "/master/iv/src/bin/mkdossier/App.h", 9,
    __D_App_parents,
    __D_App_children
);

static Dossier* __D_ArgVec_parents[] = { 0 };
static Dossier* __D_ArgVec_children[] = { 0 };

Dossier __D_ArgVec = Dossier(
    "ArgVec",
    "/master/iv/src/bin/mkdossier/ArgVec.h", 4,
    __D_ArgVec_parents,
    __D_ArgVec_children
);

static Dossier* __D_CPP_parents[] = { &__D_App, 0 };
static Dossier* __D_CPP_children[] = { 0 };

Dossier __D_CPP = Dossier(
    "CPP",
    "/master/iv/src/bin/mkdossier/Cpp.h", 9,
    __D_CPP_parents,
    __D_CPP_children
);

static Dossier* exportedClasses[] = {
    &__D_App,
    &__D_ArgVec,
    &__D_CPP,
    0
};

static int dummy = Dossier::merge(exportedClasses);

```

---

Figure 5: Initialization of dossiers

---

```

typedef class Dossier* ClassId;

extern ClassId ELLIPSE_COMP;
extern ClassId ELLIPSE_VIEW;

ClassId ELLIPSE_COMP = &__D_EllipseComp;
ClassId ELLIPSE_VIEW = &__D_EllipseView;

class EllipseComp : public GraphicComp {
public:
    EllipseComp(SF_Ellipse* = nil);

    virtual ClassId GetClassId();
    // boolean IsA(ClassId);    -- actually inherited from Component
};

boolean Component::IsA (ClassId id) {
    return GetClassId()->isA(id);
}

ClassId EllipseComp::GetClassId () { return ELLIPSE_COMP; }

ellipse.c:116:    if (tool->IsA(GRAPHIC_COMP_TOOL)) {

```

---

Figure 6: Examples of dossiers' use in Unidraw

Our current implementation generates the file “\_dossier.h” because the compiler does not know about dossier declarations. In our proposed language extension, the compiler would automatically import dossier declarations so that the application would not have to include “\_dossier.h”.

The current implementation of mkdossier calls the C preprocessor to put preprocessed copies of all the source files passed to mkdossier in a temporary directory. The preprocessor strips comments, includes header files, and expands macros so mkdossier can see the same code the C++ compiler sees when it compiles the source files. When mkdossier scans a preprocessed file, it conducts a regular expression search for the keyword “class” followed by text that looks like a class definition. When mkdossier finds a match, it extracts information about that class from its definition. Once mkdossier has scanned all of the preprocessed files, it writes the collected class information to the files “\_dossier.h” and “\_dossier.c”.

Currently we explicitly tell mkdossier which dossiers it should define so we can run mkdossier on a library's source files and include the compiled dossiers in the library. We can then run mkdossier on an application's source files without mkdossier generating duplicates of the library's dossiers. Specifying which dossiers to generate is inconvenient; we are therefore modifying mkdossier to output a dossier for a class only if at least one of the source files defines a non-inlined member function of that class. AT&T cfront 2.0 already uses a similar heuristic to decide when to generate a virtual function table.

## 4 Experience

We used the Unidraw library [4] as a test bed because it already defines symbolic class identifiers for most of the classes in the library. We changed the typedef ClassId from “unsigned int” to “class Dossier\*” and the symbolic class identifiers from integer constants to variables.

This change allowed us to use most of the library code unchanged, including all the virtual "GetClassId" member functions. We only had to replace a couple of switch statements by if statements. We replaced the virtual "IsA" member functions, which were redefined in every class, with non-virtual "IsA" member functions defined only in the base classes. The application "drawing" that uses the Unidraw library ran without error after we made these changes, thereby demonstrating that our approach is a practical way to give an application runtime access to type information. Figure 6 shows example code fragments from Unidraw that illustrate how we changed Unidraw classes to use dossiers.

We considered automatically generating a "GetClassId"-like function for existing class types. We could have written a tool to produce new header files with the function declaration added and modified mkdossier to produce the function definitions along with the dossiers. However, we decided that manually adding a single function definition in every class was not a significant problem. What is critical is that mkdossier automatically generate the type information so that the programmer does not have to define it.

## 5 Language extension

If a programmer wants to use class types developed externally, it is inconvenient to modify someone else's header files. A simple language extension would eliminate the need to change class definitions and also make it possible to obtain type information for typedef names in addition to class types.

We propose that a read-only member "dossier" of type Dossier\* be predefined for all user-defined types. By making "dossier" predefined instead of reserved, we avoid disturbing the behavior of any existing code. The programmer would use the syntax `typename::dossier` to access information for a class type or typedef and the syntax `object.dossier` or `object->dossier` to access information for a class object.

Applications need not import the Dossier interface to use a class object's "dossier" member. An application could compare the value of `object->dossier` with the value of `typename::dossier` to identify a class object's data type. Including "dossier.h" makes additional information about the class object's data type available.

Additionally, we propose that the compiler treat `object->dossier` as equivalent to `object.dossier` if the class does not have a virtual table and equivalent to a virtual function call if the class has a virtual table. For such classes, `object->dossier` will return the dossier associated with the class object's dynamic data type even if the compiler cannot determine the type statically. We call the "dossier" member of such classes a "virtual member variable." One possible implementation would generate a unique virtual table for every class and place the address of the class's dossier in the first slot of the virtual table.

An analysis of the Unidraw library revealed that only two out of 115 Unidraw classes would have shared their parent's virtual table if they had not defined a virtual "GetClassId" member function. We expect that requiring every class to have a unique virtual table whether or not it could have shared its parent's virtual table will cause only a small increase in the size of executables.

## 6 Future Work

Mkdossier must rescan all the source files whenever class information changes. If mkdossier could rescan only the source files that the compiler has to recompile, mkdossier would run faster. What we need is a way for mkdossier to remember the type information it collected last time.

We can make the previously collected type information available to mkdossier by compiling and linking the file “\_dossier.c” into the mkdossier executable after each run of mkdossier. To be practical, this approach requires the availability of shared libraries so that we can minimize the disk space occupied by many instances of mkdossier. Shared libraries allow us to store only the information that actually differs among all of the instances.

Alternatively, we could store the type information outside of mkdossier in an external file or in a database server. When mkdossier starts up, it reads the previous type information from the external file or the database server. When mkdossier shuts down, it writes the updated type information to the external file or database server. We plan to investigate which method would be the best method to let mkdossier scan files incrementally.

## 7 Summary

We have defined Dossier, an interface for accessing type information at runtime. We have implemented mkdossier, a tool that generates a C++ source file containing the type information. To demonstrate the practicality of our approach, we modified the Unidraw library and a Unidraw-based application to use dossier information. Finally, we proposed a simple extension to C++ that would provide a uniform method to access information for classes and typedefs.

## References

- [1] Keith E. Gorlen. An object-oriented class library for C++ programs. In *Proceedings of the USENIX C++ Workshop*, pages 181–207, Santa Fe, NM, November 1987.
- [2] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *Computer*, 22(2):8–22, February 1989.
- [3] Joel McCormack, Paul Asente, and Ralph R. Swick. *X Toolkit Intrinsics—C Language Interface*. Digital Equipment Corporation, March 1988. Part of the documentation provided with the X Window System.
- [4] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 158–167, Williamsburg, VA, November 1989.
- [5] Andre Weinand, Erich Gamma, and Rudolf Marty. ET++—An object-oriented application framework in C++. In *ACM OOPSLA '88 Conference Proceedings*, pages 46–57, San Diego, CA, September 1988.



# **Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling, and Garbage Collection**

Robert Seliger

Hewlett-Packard Clinical Information Systems

175 Wyman Street

Waltham, Massachusetts 02254

(617)-890-6300

robs@hpwall.hp.com

## **ABSTRACT**

Extended-C++ is a superset of the C++ programming language that offers integrated support for remote procedure call (RPC), concurrency, exception handling, and garbage collection. The language was developed in order to facilitate the implementation of a clinical information system for use in hospital critical care environments. The requirements for the information system included performance, dependability, and cost goals that could best be met by employing a distributed architecture. In addition, it was decided that the engineering of the system would benefit if it was object-oriented in both design and implementation. As described in this paper, the result of combining C++ support for object-oriented programming with the extensions offered by Extended-C++ provide a unified tool for writing distributed object-oriented programs.

Extended-C++ is implemented by a translator that generates C++ code from Extended-C++ code, and a run-time library. The process for compiling, linking, and debugging Extended-C++ programs is almost identical to the process used for standard C++ programs. The language currently runs on Hewlett-Packard series 680X0-based workstations, and it will be ported to Hewlett-Packard RISC-based processors in the near future.

Extended-C++ has been in use since 1987. The development team has written more than 250,000 lines of Extended-C++ code to implement the programs that comprise the information management system.

## **1. Introduction**

Extended-C++ is a superset of the C++ programming language that offers integrated support for remote procedure call (RPC), concurrency, exception handling, and garbage collection. The language was developed in order to facilitate the implementation of a clinical information system for use in hospital critical care environments. The requirements for the information system included performance, dependability, and cost goals that could best be met by employing a distributed architecture. In addition, it was decided that the engineering of the system would benefit if it was object-oriented in both design and implementation. As described in this paper, the result of combining C++ support for object-oriented programming with the extensions offered by Extended-C++ provide a unified tool for writing distributed object-oriented programs.

Extending an existing programming language to add linguistic support for specialized functionality is not a new idea, with C++ being a recent example. In 1986, when faced with the challenge of how to build a distributed hospital clinical information management system, the decision was made to extend an existing programming language to support the development of distributed programs. We

specifically wanted to avoid trying to build our system using an ad-hoc collection of unrelated software components. Instead, we wanted to unify the basic components that would allow programs to communicate so that writing distributed object-oriented programs would be a relatively straightforward extension to writing stand-alone object-oriented programs. C++ was chosen as the language to extend because of its support for object-oriented programming and because of its C ancestry (which enabled us to use many standard C tools).

A major concern was that we wanted most of the system's developers to spend their time developing the system, not the primitives upon which the system was to be built. Although HP-UX, Hewlett-Packard's version of UNIX<sup>1</sup>, had been chosen as the operating system, the facilities for inter-process communication were either low-level (e.g., UDP, TCP) or required additional components, tools, and mechanisms (e.g., SUN NFS<sup>2</sup>). A significant influence was the author's experience with the programming language Argus [18]. Argus was developed at MIT as a language to support the development of robust distributed programs [12]. Although the author's experience demonstrated that even with Argus, developing robust distributed systems was still difficult [8], the more profound observation was that one could develop complex distributed systems without having to learn or know much about the underlying operating system or mechanisms upon which the system was built. Consequently, an essential goal for Extended-C++ was to capture the essence of Argus (less such relatively exotic mechanisms as Argus' support for programmatic atomic transactions).

The remainder of this paper is organized into ten sections: Previous Work, Constraints, Extended-C++ Support for Exception Handling, Extended-C++ Support for Concurrency, Extended-C++ Support for Garbage Collection, Extended-C++ Support for Remote Procedure Call, the Extended-C++ Library, Implementation, and Conclusions.

## *2. Relation to Previous Work*

Linguistic support for the "specialized" functionality offered by Extended-C++ is not new. Languages such as Ada [3], CLU [13], and ML [14] support exception handling. Languages such as Concurrent-C [6], SR [1], Argus, Mesa [15], and even Algol 68 [20] support concurrency. Most Lisp [19] and Smalltalk [7] implementations support automatic memory reclamation. Languages such as Argus, Distributed Smalltalk [4], Emerald [5], and SR support distributed programming.

Extended-C++ borrows the ideas from the languages mentioned above and adds some novel constructs as well. Specific influences were Argus' models for exception handling (which is similar to exception handling in CLU), light-weight processes [11], and transmitting abstract data types [9], Mesa monitors [10], and Concurrent-C processes.

## *3. Constraints*

Most of the constraints upon Extended-C++ were dictated from the decision that Extended-C++ be implemented as a translator that produced standard C++ code from Extended-C++ code, and a run-time library that would be linked with all Extended-C++ programs. Modifications to the C++ translator or to the HP-UX kernel were not allowed. In addition, Extended-C++ was to be a superset of C++; the syntax and semantics of the C++ part of Extended-C++ were to be identical to standard C++.

---

1. UNIX is a registered trademark of AT&T.

2. NFS is a trademark of Sun Microsystems, Inc.

Additional constraints included:

- The ability to link existing C and assembly code with Extended-C++ programs was necessary.
- The performance and memory costs of a particular construct should not exceed the utility of the construct.
- Where possible, the language should expose the costs of a construct.
- Where possible, the language should not impose a cost for not using a particular construct.

In addition, Extended-C++ was not constrained to be generally portable or interoperable across diversely heterogeneous machines or operating systems. It would suffice if the task of porting Extended-C++ to different machines was non-trivial, and it did not matter whether Extended-C++ could be readily ported to non-HP-UX versions of UNIX. Also, it did not matter if existing C++ code had to be recompiled in order to be linked with Extended-C++ programs.

#### 4. Exception Handling

Exception handling in Extended-C++ offers a mechanism where:

- programs can uniformly deal with errors
- ad-hoc conventions about which return values represent "good" values and which return values represent "bad" values can be avoided
- a called function can force its caller to "pay attention" to an exceptional situation; the caller cannot ignore the exceptional situation without repercussions
- a calling function can decide whether to "pay attention" or to "ignore" potential exceptional situations indicated by called functions
- the performance of programs can be improved by reducing the amount of code required to test for exceptional situations
- writing modular software can be facilitated by making it easier to separate code for "normal" processing from code for "exceptional" processing.

##### 4.1 Exception Handling Constructs

Extended-C++ exception handling is comprised of four basic constructs:

- exception names which identify specific exceptions
- exception declarators which itemize the exceptions that a function can raise
- statements that can be used to raise an exception
- statements that can be used to handle exceptions.

**4.1.1 Exception Names** Exception names are identifiers. An exception name is always global in scope and all uses of a particular exception name identify the same exception.

There is no need for the programmer to specially declare exception names (other than within function prototypes) or to assign or manage exception identifiers.

**4.1.2 Exception Declarators** The exceptions that can be raised by a function are listed in the function's declarator, immediately following the function's argument declaration list. Although exceptions cannot be used to overload functions, they are considered part of a function's prototype. A function does not have to raise any of the exceptions that it declares. In the following example, the member function *PtrArray::operator[]* which raises the exception *OutOfBounds* is declared:

```
class PtrArray
{
public:
    void *operator[] (int index) raises (OutOfBounds);
    // other member functions
private:
    void **array; // pointer to array of void *
    int num;      // number of elements in array
};
```

**4.1.3 Exception Raising Statements** An exception can be raised using the **raise** or **yield** statement. When the **raise** statement is executed, the executing function returns with the identified exception raised. The **yield** statement is similar to the **raise** statement, but is used for raising an exception that is to be handled by the exception raising function. Executing a **yield** statement is the same as calling a function which raises the exception identified in the **yield** statement.

Any automatic class objects that go out of scope due to the raising of the exception are destroyed before an attempt is made to handle the exception.

An optional exceptional return value can be supplied when the exception is raised. This value must be an expression that evaluates to a **char \***. If an exceptional return value is not specified, then the value **NULL** is implicitly returned<sup>3</sup>.

In the following example, the member function *PtrArray::operator[]* raises the exception *OutOfBounds* (and provides an exceptional return value) when given a value for the argument *index* that is not within the bounds of the *PtrArray*:

```
void *PtrArray::operator [] (int index) raises (OutOfBounds)
{
    if (index < 0 || index > num - 1) {
        const char *cv = "Max index is %d";
        char *v = new char[sizeof(cmsg) + 10];
        sprintf(msg, v, num);
        raise OutOfBounds (v);
    }
    return array[index];
}
```

**4.1.4 Exception Handling Statements** An exception handling statement associates exception handlers with a block, including the block that defines the body of a function. When an exception is caught by an exception handling statement, any automatic class objects that have gone out of scope are destroyed and the values of local variables are restored to what they were before the exception was

3. As with any return value, it is the programmer's responsibility to ensure that the memory pointed to by the exception return value is within the scope of any function that might handle the exception. In addition, if the memory occupied by the object whose address is returned as an exceptional return value is dynamically allocated, then the deallocation of the object can either be managed by the user or can be left for the garbage collector to reclaim.



raised.<sup>4</sup> The exception handling statement is then examined for a handler for the exception.

There are two styles of exception handling statements. The **except** statement allows different actions to be taken for different exceptions, and the **reraise** statement allows exceptions to be explicitly propagated from one function to another.

The **except** statement is a switch-like statement composed of handler clauses. Each clause consists of the keyword **when**, the list of the exceptions that the clause will handle, and an action to be taken in order to handle the exception. The special handler clause **others** handles any exception. A handler clause action can be any statement, including nested exception handling statements.

An **except** statement and the block that it is associated with define disjoint scopes, but both scopes are enclosed by the same scope. Once an exception handler clause has been executed, program execution resumes at the statement immediately following the exception handling statement (unless otherwise directed by a statement that alters program flow of control, e.g., **return**, **goto**).

The **reraise** statement is comprised of the keyword **reraise** followed by the names of the exceptions to be propagated without an intervening action by the propagating function. Using an asterisk (i.e., **reraise \***) in lieu of a list of exception names informs the compiler to generate code to catch and **reraise** any of the exceptions that the function can raise.

There are no constraints as to which exceptions can be itemized within an exception handling statement. Conversely, an exception handling statement does not have to be implemented to handle every exception that can possibly be raised within the block it is associated with. In the following example, function *f* handles the exception *OutOfBounds* by printing the string pointed to by the exceptional return value and returning a NULL pointer value:

```
void *f(PtrArray& a)
{
    void *p = a[i];
    // do something with p
    return p;
}
except when OutOfBounds (char *v): {
    if (v) {
        print("%s", v);
        delete v;
    }
    return NULL;
}
end;
```

**4.1.5 Propagating an Exception and Unhandled Exceptions** If a handler for an exception is not found within the exception handling statement being searched, then the exception is propagated to the closest enclosing block that has an associated exception handling statement. As an exception is propagated, any automatic class objects that have gone out of scope due to the exiting of a block are destroyed. This process continues until the exception is either handled or there are no more

4. This is a subtle point, but it is distinctly different from the semantics of mechanisms like the UNIX *setjmp/longjmp* functions which restore the values of local variables to the values that they had when the function *setjmp* was called. This behavior was considered too confusing for exception handling and would make it impossible to do something as simple as determine the value of an iterator variable for a loop exited due to an exception.

enclosing blocks with associated exception handling statements, at which point the exception becomes an unhandled exception <sup>5</sup>.

When an unhandled exception occurs, the optional user-defined function *unhexception* is called. If *unhexception* does not explicitly exit or abort the program, then when this function returns, the program will be terminated immediately *without* the execution of static destructors. Alternatively, if *unhexception* explicitly calls the function *exit*, then static destructors will be executed <sup>6</sup>.

If a user-defined version of *unhexception* is not provided, then a default version is called. The default version reports information about the unhandled exception and then terminates without executing static destructors.

It is possible that a function called directly or indirectly by *unhexception* will raise an exception that is not handled. When this occurs, the situation is reported and the program is terminated immediately.

**4.1.6 Exceptions and Constructors and Destructors** Constructors and destructors can raise and handle exceptions like any function, but there are subtle differences. If the constructor for a derived class raises an exception, then the object is considered to have been constructed, so it can be safely destroyed. In contrast, if the constructor for a base class or class member raises an exception, then the result is a partially constructed object. Even if a derived class constructor handles such an exception (using an exception handling statement associated with the body of the derived class constructor), the constructor will only be able to return with a constructor-failed exception raised. (The name of a constructor-failed exception is of the form *<class-name>CtorFailed*.) It is advised that constructor-failed exceptions not be handled because partially constructed objects are generally unreliable.

If the destructor for a derived class raises an exception, the exception is deferred until after the object's base and class members have been destroyed. When the exception is finally raised, the object is considered to have been destroyed. In contrast, if the destructor for a base class or class member raises an exception and a derived class destructor handles such an exception (using an exception handling statement associated with the body of the derived class destructor), then the object is partially destroyed. Even if a derived class destructor handles such an exception (using an exception handling statement associated with the body of the derived class destructor), the destructor will only be able to return with a destructor-failed exception raised. (The name of a destructor-failed exception is of the form *<class-name>DtorFailed*.) It is advised that destructor-failed exceptions not be handled because partially destroyed objects are generally unreliable.

## 5. Concurrency

Extended-C++ support for concurrency provides constructs that allow programmers to define light-weight processes (or *threads*), control the lifetimes of threads, and synchronize and coordinate threads. Threads can be dynamically created and destroyed, and there is no language-imposed limit

5. The fact that exceptions are not implicitly propagated across function boundaries is consistent with the fact that the exceptions that a function can raise are part of the function's interface. A function's interface would be violated if exceptions not declared in the function's interface could be seen by callers of the function.

6. And, on HP-UX, if the function *abort* is called, static destructors will *not* be executed, but a core dump will be produced.

on the number of threads that can execute in a program at a particular time. In addition, all threads execute in a common address space, thereby providing a program's threads with common access to all text and data within the program. Thread stacks are an exception; a thread can only access its own stack.

Extended-C++ support for concurrency is no more expensive than that provided by the host operating system, and in the case of UNIX, Extended-C++ support for concurrency can be significantly less expensive. Extended-C++ support for concurrency offers an alternative to "heavy-weight" processes, and by the nature of Extended-C++ concurrency, problems that would otherwise be difficult to solve by using the host operating system or by writing traditional, non-concurrent programs are simplified.

### 5.1 Main Thread

Every Extended-C++ program has at least one thread. This thread implicitly begins execution at the start of the user function *main* and implicitly terminates when the function *main* completes. This initial thread can explicitly create other threads.

The termination of the initial thread does not cause an Extended-C++ program to terminate. This can only be achieved by calling one of the standard C library functions *exit*, *\_exit*, or *abort*.

### 5.2 Threadable Functions

A function declared with the qualifier **threadable** can be used to begin the execution of a thread. Member and non-member functions can be **threadable**, and **threadable** functions can be overloaded<sup>7</sup> and virtual, and can accept any number and type of arguments. Pointers to **threadable** functions can be declared. Constraints on **threadable** functions are: they must be **void** valued, they cannot raise exceptions, and they cannot accept a variable number of arguments. Also, operators, constructors, and destructors cannot be declared to be **threadable**. A **threadable** function can only be called when the call is the operand for the **threadit** operator, as described in the next section.

### 5.3 Threadit Operator

The **threadit** operator defines the point in a program at which a new thread is to be created or *spawned*. The operand for this operator must be an expression that evaluates to a call to a **threadable** function. A statement comprised of a **threadit** operator and operand is referred to as a **threadit** statement.

When a **threadit** statement is executed, a new thread is spawned which begins by executing the **threadable** function that the **threadable** operator was applied to. The spawning thread then resumes execution with the statement immediately following the **threadit** statement. The new thread executes concurrently with the thread that spawned it and with all other currently executing threads.

There is no specification as to when a newly spawned thread actually begins execution or when the spawning thread resumes execution. Once a thread begins execution, it may voluntarily preempt itself by blocking (e.g., it can go to sleep) or it may be periodically involuntarily preempted. Attempts to involuntarily preempt a thread are ignored if the thread is currently executing a *non-preemptable* function. Modules which contain non-preemptable functions are identified when a

7. The keyword **threadable** also overloads a function's name such that non-**threadable** and **threadable** functions with the same scope and prototype can have the same function name.

program is linked <sup>8</sup>.

**5.3.1 Thread Arguments and Stack** When a thread is spawned, an execution stack is created for it which contains the arguments to the **threadable** function that initiates the thread's execution. The arguments to a **threadable** function can be passed by value, address, or reference. The appropriate constructors are used to create copies of class objects passed by value (and any temporary class objects produced are destroyed). There is no attempt to ensure that pointer and reference arguments are to objects that are accessible to the new thread. In particular, threads are at risk if they pass pointers or references to stack-allocated objects.

Thread stacks can grow and shrink and the size of a thread's stack is limited only by the memory configuration of the host operating system.

**5.3.2 Shared Data** Global, static, and dynamically allocated objects can be accessed by any thread. Coordinating access to shared data is the responsibility of the programmer. The coordination is achieved by using Extended-C++ synchronization primitives or by limiting knowledge about the existence of a particular object to a single thread <sup>9</sup>.

**5.3.3 Termination** A thread terminates by returning from a **threadable** function. When a thread terminates, all automatic class objects within the scope of the **threadable** function that have been created by the function, but that have not yet been destroyed, are destroyed. A thread can also be terminated by executing the **terminate** statement, but in this case, automatic class objects that have been created by the thread, but that have not yet been destroyed, are *not* destroyed.

**5.3.4 Threadable Functions Example** In the following example, program execution starts with the implicitly created first thread that executes the function *main*. The **for** loop then spawns a total of three threads. In the absence of explicit synchronization, the scheduling of the three threads and the initial thread, relative to each other, is not specified.

```
void printvals(int v1, int v2) threadable { print("[%d.%d]", v1, v2); }
```

```
main()
{
    print("[begin]");
    for(int i = 0; i < 3; ++i)
        threadit printvals(i, i);
    print("[end]");
}
```

An example of one possible output of the program is: [begin][2.2][0.0][end] [1.1].

---

8. The ability to identify that a module contains non-preemptable functions made it easy to solve most of the problems encountered when using non-reentrant functions, such as those in the standard C library, with preemptively scheduled threads.

9. One exception is the global variable *errno*, defined as part of the standard C library. The value of this variable is initially 0, and at any point in time its value will be thread-specific; in a sense, *errno* is a thread-specific global variable. This means that for a particular thread, the value of *errno* will only be affected by a function executed by the thread. In addition, the thread will only "see" *errno* values that it has set.



#### 5.4 Subthreadit Operator

The **subthreadit** operator is similar to the **threadit** operator except that the spawning thread is blocked until *after* the spawned thread terminates. The **subthreadit** operator is intended for use with the **coenter** statement and loops, as discussed in the next section.

#### 5.5 Coenter Statement and Loops

The **coenter** statement and the various C++ loop constructs allow a thread to spawn a collection of threads. The arguments for all of the threads in the collection are evaluated before any of the threads begins execution and before the spawning thread resumes execution.

A **coenter** statement can only contain, in any quantity and ordering, **threadit** and **subthreadit** statements. In order for a loop construct to be used to spawn threads as described, the iterated statement must be a single **threadit** or **subthreadit** statement.

If one or more of the statements contained within a **coenter** statement is a **subthreadit** statement, or if the statement iterated in a loop is a **subthreadit** statement, then the spawning thread blocks until *all* of the threads spawned by the **subthreadit** statements have terminated.

In the following example, the threads spawned when the **coenter** statement is executed will each "see" the same value for *i*. In addition, the thread that executes the function *main* will not resume execution until the two threads spawned when the **subthreadit** statements are executed have terminated.

```
main()
{
    int i = 3;
    print("begin");
    coenter {
        subthreadit printvals(0, i);
        subthreadit printvals(1, i);
        threadit printvals(2, i);
    }
    ++i;
    print("end");
}
```

The output produced by this program will always begin with the printing of [begin], and the printing of [0.3] and [1.3] will always precede the printing of [end]. However, the relative ordering of any other output cannot be determined.

#### 5.6 Region Statement

A **region** statement can only be executed by one thread at a time. A thread that attempts to enter a region that is already occupied is blocked on the tail of a FIFO queue of threads waiting to enter the region. The **region** statement provides a mechanism that allows programmers to ensure that a thread gains exclusive access to text or data. A **region** statement can identify a unique region, distinct from all other regions in a program, or different **region** statements can be mapped to the same region. In the latter case, a **const void \*** valued expression can be used to identify the region that a particular **region** statement represents.

A thread exits a region by reaching the end of the **region** statement, or by passing control out of the region statement via a flow-of-control affecting statement, such as a **return** or **raise** statement. An **except**, **reraise** or **reraise \*** statement associated with a **region** statement is considered to be outside

of the region.<sup>10</sup>

A thread that is already in a region can unconditionally re-enter the region any number of times. A thread is not considered to have exited a particular region until the thread exits each of the **region** statements that it executed to enter the region, or when it terminates.

### 5.7 Release and Trigger Statements

The **release** statement causes a thread to block inside of a region, thereby allowing another thread to enter the region. A thread that has executed the **release** statement is placed at the tail of a FIFO queue of threads that have also executed the **release** statement within the region. A thread that has executed the **release** statement remains blocked until another thread enters the region, executes the **trigger** statement, and then either exits the region or executes the **release** statement, thereby *triggering* the region's queue of *released* threads.

Once a region's queue of released threads is triggered, each thread on the queue is awakened one at a time. When the awakened thread exits the region or executes the **release** statement again, the next thread on the region's release queue is awakened. The triggering continues until the thread that was at the tail of the queue at the time that the **trigger** statement was executed has been awakened.

If a triggered thread in turn executes the **trigger** statement, then the region's queue is considered to have been newly triggered.

When combined as described above, the **region**, **release**, and **trigger** statements implement mutual exclusion, prevent the need for threads to busy-wait, and provide a mechanism for an active thread to explicitly awaken a blocked thread.

In the following example, a thread that attempts to "pop" an empty *Stack* will block until another thread "pushes" something on the *Stack*:

```
void *Stack::pop ()
{
    region(this) {
        if (mySize == 0) release;
        return myItems[mySize-- - 1];
    }
}

void Stack::push (void *item) raises (Full)
{
    region(this) {
        if (mySize == myCapacity) raise Full;
        myItems[++mySize - 1] = item;
        trigger;
    }
}
```

10. The ability of the compiler to determine the many ways in which a region can be exited illustrates the power of using language primitives and semantic analysis, instead of function calls and a reliance on the programmer to remember and follow all the rules.

## 5.8 Condition Variables

Using variables of the built-in class type **condition**, threads can coordinate their actions against some logical condition. The **condition** type provides a mechanism for blocking a thread on a queue of threads until a user-defined event transpires, as indicated by an explicit user-defined action.

The two basic categories of operations that can be performed on **condition** variables are notification and waiting. Notices can be buffered or unbuffered, and waits can be unlimited in duration or can have time-outs specified.

The following example illustrates the use of the built-in type **condition**. By using the **condition** variable *C*, the **threadable** functions *f1* and *f2* can be synchronized:

```
condition C;

void f1() threadable { print("[Hello]"); C.notify() }
void f2() threadable { C.wait(); print("[Good-bye]"); }

main()
{
    threadit f1();
    threadit f2();
}
```

The output produced is always: [Hello][Good-bye].

## 6. Garbage Collection

Like I/O, garbage collectors are generally not considered to be part of a language. However, they are nevertheless described because they are so intimately related. This situation is perhaps even more true for the Extended-C++ garbage collector because its behavior and functionality are heavily influenced by C++ semantics. For the most part, although the description of the garbage collector provided in this section may appear to refer to a particular implementation, it is in fact a description of the set of requirements that would characterize any Extended-C++ garbage collector implementation.

The Extended-C++ garbage collector offers the traditional functionality of reclaiming memory no longer in use. The garbage collector is non-traditional in that it can be used in environments that offer no special support for garbage collection, and can be used with both C++ class and scalar objects.

In Extended-C++, any object allocated using the global operator **new** is registered with the garbage collector. These objects are referred to as *garbage collectible*. When an object is registered, it is both a candidate for future garbage collection and considered a source of references to other objects registered with the garbage collector. When the garbage collector decides to perform a collection it determines for each of the objects registered with it whether the object is reachable from a reference in a thread's stack, within a thread's registers, from a static object, or from another object registered with the garbage collector. The memory for any object which is not reachable is reclaimed. If the unreachable memory is occupied by a class object with a destructor, then the destructor is executed first.

The Extended-C++ garbage collector uses a mark-and-sweep algorithm. The execution of the mark-phase, during which the set of reachable objects is determined, cause all threads to be suspended. In contrast, the sweep phase, during which all unreachable objects are reclaimed, is

executed by a thread that runs concurrently with other threads. This organization shortens the time during which threads are suspended by a garbage collection. This is of particular importance because the act of reclaiming objects can involve arbitrary work as destructors are executed.

The garbage collector is quite flexible, as indicated by the following features:

- Although memory allocators and deallocators are provided, a collection of library functions makes it possible to employ user-defined allocators and deallocators <sup>11</sup>.
- Explicit memory management of garbage collectible objects can coexist with the garbage collector.
- The garbage collector can be enabled or disabled using a run-time option that all Extended-C++ programs accept.
- The garbage collector can be enabled in automatic mode, where it determines when to execute, or in demand-only mode, where the program instructs it when to execute.
- Using the function *gcselect*, the garbage collector can be instructed to abort a garbage collection and to try again later when I/O is possible on a particular file or device.

The following subtleties about the garbage collector behavior are important to C++ programs:

- Pointers to the interiors of objects are detected. This is most important when multiple-inheritance is used because, due to pointer casting, a program may only have a pointer to one or more of an object's base or class members, and not to the beginning of the object.
- Due to destructors, an "unreachable" object may still be reachable from another "unreachable" object that has a destructor. The garbage collector is careful to not deallocate an object until it is not reachable from *any* object. (See [2] for a detailed discussion of this problem.)

The current garbage collector implementation has the following limitations:

- It does not scan dynamically allocated memory not registered with it for references to garbage collectible objects.
- It does not deallocate cyclical data structures if all of the objects in the cycle have destructors.

## 7. Remote Procedure Call

The intent of Extended-C++ support for remote procedure call is to facilitate the development of distributed object-oriented programs that use a *fine-grain* object model of computation. In Extended-C++, remote objects can be pointed to and member functions can be invoked on the remote objects.

Remote procedure call in Extended-C++ is modeled after the familiar local function call paradigm with the major difference that a remote function call generally requires an exchange of inter-process messages between the caller's program and the callee's program <sup>12</sup>. This difference makes it difficult to implement remote procedure call semantics that are identical to local function call semantics [16].

11. The use of user-defined allocators is facilitated by the built-in operators *gesizeof* and *geoffsetof*. The former can be used to determine how much memory a garbage collectible object requires and the latter reports where the user portion of a garbage collectible object begins, relative to the memory allocated for it.

12. In this discussion, the terms *procedure* and *function* are used interchangeably.



One similarity in Extended-C++ between local and remote function calls is that the caller of a local or remote function always waits until the call returns. In addition, the syntax for remote function calls is identical to the syntax for local function calls. However, it takes longer to call a remote procedure than a local procedure and it is possible for the caller's program and the callee's program to independently fail.

### 7.1 Remotable Classes and Remotable Functions

A class that defines member functions which can be called remotely is referred to as a *remotable* class. A class is **remotable** if it defines or inherits at least one function declared as **remotable**<sup>13</sup>.

Any non-static member function, including operators, declared with the qualifier **remotable** can be called remotely. A **remotable** function can be overloaded<sup>14</sup> and virtual, and can accept arguments (including arguments with default values), return a value, and raise exceptions.

The following is an example of a class declaration with several **remotable** functions and both local and **remotable** versions of operator []:

```
class R
{
public;
    void f() const remotable;
    virtual int g(const char *cp) remotable;
    int operator[] (int i) remotable;
    int operator[] (int i);
};
```

**7.1.1 Argument Passing** The types of objects that can be used as **remotable** function arguments or return values are limited to *transmissible* types. A transmissible type can be encoded into and decoded from messages suitable for transmitting a **remotable** function call request or reply. For each transmissible type, *encoder* and *decoder* functions are defined for encoding and decoding instances of the type. Encoders and decoders are discussed in more detail later.

A **remotable** function's arguments and return value are always passed by value in the sense that a copy is always made of each argument and the return value. In addition, the following rules apply:

- Passing a pointer argument to a **remotable** function results in the object pointed to being encoded as a *shared* object. A shared object is encoded and decoded at most once per **remotable** function call.
- Passing a **remotable** function argument by value or reference results in the object being encoded as a *value* object. A value object can be encoded and decoded an unlimited number of times per **remotable** function call.
- A **remotable** function that returns a pointer to an object returns a pointer to a shared object.
- A **remotable** function that returns an object by value or reference returns a value object or reference to a value object, respectively.
- Arrays cannot be passed as **remotable** function arguments or returned as **remotable** function return values.

13. The term *remotable* is a concatenation of the terms *remote* and *invokable*.

14. The keyword **remotable** also overloads a function's name such that a local function and a **remotable** function with the same scope and prototype can have the same function name.

- Memory is dynamically allocated using the standard allocator or a class-specific allocator, as appropriate, for **remotable** function arguments or return values that are shared objects.
- Memory is allocated on the stack for **remotable** function arguments or return values that are passed by value. Memory is dynamically allocated for **remotable** function arguments or return values that are passed by reference or address.
- One or more stack-allocated copies can be made of an object passed by value as a **remotable** function argument or return value. For example, if the object is a class object *X*, then the *X::X(X&)* constructor will be used to make the copies.

**7.1.2 Execution Semantics** In order to perform a **remotable** function call, the caller's thread composes a request message which includes encoded copies of the function arguments, transmits it to the callee's program, and then blocks waiting for a reply message. Upon receipt of the new request message, the callee's program creates a new thread to service the request. This thread decomposes the request message, creates the function's arguments from the encoded arguments and executes the **remotable** function. The function's return value is then encoded into a reply message which is transmitted back to the caller's thread. The caller's thread creates a return value by decoding the reply message, and then resumes execution. The callee's thread terminates after it has successfully transmitted the reply message (or given up due to a failure).

Even though request and reply messages can be lost or duplicated, each **remotable** function call results in at most one execution of the called function. If the **remotable** function call request is not successfully transmitted, then the called function will not execute at all. If the **remotable** function call request is successfully transmitted, a subsequent failure in the callee's program may result in the incomplete execution of the called function. If the **remotable** function call request is successfully transmitted, and any subsequent failures occur after the **remotable** function call reply has been transmitted, then the **remotable** function will execute exactly once.

**7.1.3 Communication Exceptions** The built-in exceptions *RpcFailure*, *RpcUnavailable*, *RpcOldRemPtr*, and *RpcCorruptRemPtr* indicate that a **remotable** function call has either experienced a permanent communication problem, has experienced a temporary communication problem, has been attempted on a **remotable** object that no longer exists, or has been attempted using a corrupt **remotable** pointer.

The caller's thread can handle these exceptions. In contrast, the callee's thread program never sees these exceptions. This implies that communication difficulties will not prevent a **remotable** function that has been successfully called from running to completion. If the callee's thread cannot transmit a **remotable** function return value, then the thread simply terminates as it would normally. Although the **remotable** function has effectively been *orphaned* [17], it will not be aborted (due to communication problems) in the middle of its computations. From the callee's thread's perspective, there is no difference between an orphaned **remotable** function call and a **remotable** function that has not experienced communication difficulties.

**7.1.4 Remotable Class Instantiation** In order for a program to use a **remotable** class solely for the invocation of **remotable** functions, the class declaration is required and the class must be *instantiated*. Instantiating a **remotable** class causes code to be generated that allows programs to execute the class's **remotable** functions without requiring that the program be linked with the definitions of the **remotable** functions.

A **remotable** class is instantiated using the following syntax <sup>15</sup>:

```
remotable class R = 0;
```

## 7.2 Remotable Pointers

When used in the declaration of a pointer, the keyword **remotable** declares that the pointer has the *capability* to point to a **remotable** object. A **remotable** pointer can be dereferenced to call **remotable** member functions. The syntax for calling a **remotable** function through a **remotable** pointer is the same as for calling a local function through a local pointer.

The object pointed to by a **remotable** pointer can reside in any Extended-C++ program, including the program calling the **remotable** function. Independent of where the object resides, a **remotable** function call always has the same semantics.

A **remotable** pointer can be initialized from a **remotable** object within the same program by using the `@` operator. This operator is analogous to the `&` operator. The `&` operator applied to an `X` evaluates to an `X*`; the `@` operator applied to an `X` evaluates to a **remotable** `X*` if `X` is a **remotable** class, or otherwise is a compile-time error.

A **remotable** pointer can also be initialized from or assigned the value of another **remotable** pointer subject to the same C++ rules that govern the assignment and casting of pointers to local class objects. In addition, an *untyped* **remotable** pointer<sup>16</sup> can be initialized from or assigned the value of *any* type of **remotable** pointer. Conversely, a typed **remotable** pointer can be initialized from or assigned the value of an untyped **remotable** pointer, but explicit casting is required. An untyped **remotable** pointer is declared using just the keyword **remotable**.

Given that `X` is a **remotable** class with **remotable** function `f` and that `x` is an `X`, the following example illustrates the use of **remotable** pointers:

```
remotable const X *xrp = @x;    // remotable pointer to a const X
X *xp = &x;                    // local pointer a const X
xrp->f();                        // remote call to const function X::f
xp->g();                         // local call to const function X::g
const remotable *rp = xrp;      // untyped remotable pointer
((remotable const X *)rp)->f(); // casting an untyped remotable pointer
```

## 7.3 Transmissible Types

A type that can be used as a **remotable** function argument type or return value type is referred to as a *transmissible* type. All of the fundamental types are implicitly transmissible, as is any enumeration and the value of any **remotable** pointer. The value of a local pointer is not transmissible, and a class is transmissible only if an *encoder* and *decoder* is explicitly declared and defined for the class. Using its encoder and decoder, a transmissible object can be encoded into and decoded from messages suitable for transmitting a **remotable** function call request or reply.

An transmissible object can be encoded and decoded as a *shared* or *value* object. A shared object is encoded and decoded at most once per **remotable** function call; a value object can be encoded and decoded an unlimited number of times per **remotable** function. By controlling whether objects are encoded and decoded as shared or value, the depth of encoding and degree of sharing of **remotable** function arguments and return values can be controlled. This functionality allows programmers to use complex structures like a circular linked list as a **remotable** function argument or return value

15. This syntax was borrowed from the C++ syntax for defining a pure virtual function.

16. An untyped **remotable** pointer should not be confused with an untyped pointer (i.e., `void*`), the latter of which can be assigned the value of *any* type of **remotable** or non-**remotable** pointer.

with almost the same ease as a single character.

**7.3.1 Encoders** An encoder is a **void** valued **const** virtual member function that is declared as `<X` for a class *X*. Encoders cannot be called explicitly; they are called implicitly in order to encode a **remotable** function's argument or return value. They are also called as the result of applying the **encode** operator, as described in the next section.

An encoder can selectively encode an object's data members. Syntax similar to that for constructor initializers is used to define an encoder that encodes base classes or class members, and the body of an encoder can do arbitrary work including the encoding of non-member data. Any base class or class member that is to be encoded is listed by name in the encoder initializer. Each name must be followed by a pair of parentheses. The actual order of encoding corresponds to the order in which the base classes and class members are declared in the class declaration, with base classes encoded before class members, independent of the order in which the names are listed.

In the following example, the order of encoding of class *D* is *A*, *C*, *a*, *b* (and note that base class *B* and class member *c* are not encoded):

```
class D : virtual private A, public B, public C
{
    <D(); // encoder declaration
    A a;
    B b;
    C c;
    // other members
};

D::<D() : C(), A(), b(), a()
{
    // body of encoder
}
```

**7.3.2 The Encode Operator** The **encode** operator is a **void** valued unary operator that, when applied to a **transmissible** object, results in the invocation of an encoder to encode the object into the current **remotable** function caller's request or callee's reply message. The **encode** operator cannot be used outside the scope of an encoder, and can only be applied to **transmissible** objects, pointers to **transmissible** objects, **transmissible** object references, arrays of **transmissible** objects, and arrays of pointers to **transmissible** objects.

Applying the **encode** operator to a fundamental type results in the execution of the encoder implicitly defined for the fundamental type. Applying the **encode** operator to a class object results in the execution of the class's encoder. Applying the **encode** operator to a pointer or reference to a class object results in the execution of the encoder defined for the object's actual (i.e., derived) class.

When the operand is a pointer to a **transmissible** object, the object pointed to is encoded as a shared object. The value of the pointer to the object is not encoded unless the pointer value is **NULL**, in which case this fact is encoded in lieu of an object. When the operand is a **transmissible** object or **transmissible** object reference, the object is encoded as a value object.

Arrays of **transmissible** objects can be encoded by specifying the number of elements to be encoded. If the array is multi-dimensional, then the number of elements in each dimension is separately specified, as illustrated in the following example of encoding a 10x20 array of **ints**:



```
int array10by20[10][20];
encode [10][20] array10by20;
```

An array is always encoded as a shared object, but each object in an array of **transmissible** objects is encoded as a value object, and each object pointed to by an array of pointers to **transmissible** objects is encoded as a shared object.

**7.3.3 Encoder Example** The following example of an encoder illustrates how a circular linked list can be encoded:

```
struct LinkItem
{
    <LinkItem();           // encoder declaration
    LinkItem *prev;        // pointer to previous LinkItem
    LinkItem *next;        // pointer to next LinkItem
    // other members
};

LinkItem::<LinkItem()
{
    encode prev; // encodes prev LinkItem if not already encoded
    encode next; // encodes next LinkItem if not already encoded
    // encode other members
}

struct Service
{
    void f(LinkItem *head) remotable;
};
```

In the example, *LinkItems* are encoded as shared objects. When the **remotable** function *Service::f* is called, the *LinkItem* pointed to by *head* is encoded into the caller's request message as a shared object. When this object is encoded, it in turn encodes the *LinkItems* pointed to by its members *prev* and *next* as shared objects. The encoding recurses until a *LinkItem* that has already been encoded is encountered. When this happens, an encoder is not executed, and the recursion terminates. When the encoding completes, each of the *LinkItems* in the list will have been encoded once.

**7.3.4. Decoders** A decoder is a **static** member function that is declared as  $>X$  for a class *X*. As with encoders, decoders cannot be called explicitly; they are called implicitly in order to decode a **remotable** function's argument or return value. They are also called as the result of using the **decode** operator, as described in the next section.

A decoder creates a new object by selectively decoding the object's data members. The body of a decoder can do arbitrary work including the decoding of non-member data. An object created by a decoder is always initialized, even if no data is actually decoded.

Syntax similar to that for constructor initializers is used to define a decoder that decodes base classes or class members. Any base class or class member that is to be decoded is listed by name in the decoder initializer. Each name must be followed by a pair of parentheses. A base class or class member not listed in the initializer is initialized using the appropriate argument-less constructor<sup>17</sup>, and the appropriate constructor is called for any initializer that specifies arguments. Independent of

the order in which the names are (or are not) listed, the actual order of decoding or initialization corresponds to the order in which the base classes and class members are declared in the class declaration, with base classes decoded before class members.

In the following example, the order of decoding and initialization of class *X* is *A*, *B*, *C*, *a*, *b*, *c*. (Note that base class *B* and class member *c* are not decoded but are initialized, *B* is initialized using an argument-less constructor, and *c* is initialized using a constructor that requires an argument.)

```
class D : virtual private A, public B, public C
{
    >D(); // decoder declaration
    A a;
    B b;
    C c;
    // other members
};

A::>A() : C(), c(1), A(), b(), a()
{
    // body of decoder
}
```

**7.3.5 The Decode Operator** The **decode** operator is a unary operator that, when applied to an operand that is a *type-declarator*, results in the invocation of a decoder to decode an object from the current **remotable** function caller's request or callee's reply message. The **decode** operator cannot be used outside the scope of a decoder and can only be applied to a *type-declarator* that is for:

- the name of a fundamental type or **transmissible** class - an object of the type named is decoded using the type's decoder as a value object
- a pointer to a fundamental type or **transmissible** class - an object of the type named is decoded using the type's decoder as a shared object
- a **remotable** pointer - the value of a **remotable** pointer of the type named is decoded using the decoder implicitly defined for **remotable** pointers.

When the type-name portion of the *type-declarator* is the name of a class, the decoder defined for the actual (i.e., derived) class of encoded object is executed. For example, if class *B* is derived from class *A*, and the object currently being decoded was produced by encoding a *B*, then the following statement will decode a *B*, but evaluate to an *A* \*:

**decode A**

If an object decoded as a shared object has already been decoded, then the **decode** expression evaluates to a pointer to the previously decoded object; a decoder is not executed. If an object decoded as a shared object was originally encoded from a NULL pointer, then the **decode** expression evaluates to a NULL pointer. Otherwise a decoder is executed to create a new object from the decoded data and the **decode** expression evaluates to a pointer to the new object. If an object is decoded as a value object, then a new object is always created from the decoded data by executing a decoder, and the **decode** expression evaluates to a pointer to the new object.

17. In this regard, decoders differ from constructors. Listing a member *m* in a constructor initializer as *m()* is equivalent to not listing it at all; both styles will cause an argument-less constructor to be used to initialize the member. However, in the case of a decoder, not listing the member *m* and listing it as *m()* have two different consequences.

When an object is created by application of the **decode** operator, memory will be dynamically allocated using the standard allocator for fundamental types and any class-specific allocator for classes. Additionally, a decoded object can be placed at a particular memory address using **decode** operator *placement* syntax. The address at which to place a decoded object can be specified by following the **decode** operator with a parenthetically enclosed expression that evaluates to the desired address. In the following example, the decoded **char** is placed at the address occupied by the local **char** *c* and the **decode** operator evaluates to the address of *c*:

```
char c;
decode (&c) char;
```

Arrays of **transmissible** objects can be decoded by specifying the number of elements to be decoded. If the array is multi-dimensional, then the number of elements in each dimension is separately specified, as shown in the following example of decoding a 10x20 array of **ints**:

```
decode int[10][20];
```

An array is always decoded as a shared object, but each object in an array of **transmissible** objects is decoded as a value object, and each object pointed to by an array of pointers to **transmissible** objects is decoded as a shared object.

**7.3.6 Decoder Example** The following decoder example, based upon the example of an encoder shown previously, illustrates how a circular linked list can be decoded:

```
LinkedItem::>LinkedItem()
{
    prev = decode LinkedItem *; // decodes prev LinkedItem if not already decoded
    next = decode LinkedItem *; // decodes next LinkedItem if not already decoded
    // decode other members
}
```

In the example, *LinkedItems* are decoded as shared objects. When a program receives a **remotable** function call request message for *Service::f*, a *LinkedItem* is implicitly decoded as a shared object and the address of the object is assigned to the function's argument *head*. When the decoder for the *LinkedItem* pointed to by *head* is executed, the encoded *LinkedItems* pointed to by the caller's copy of *\*head*'s members *prev* and *next* are decoded by executing their decoders. The decoding recurses until a *LinkedItem* that has already been decoded is encountered. When this happens, a decoder is not executed; instead, the address that the object was decoded to is returned, and the recursion terminates. When the decoding completes, each of the encoded *LinkedItems* will have been decoded once.

**7.3.7 Encoder and Decoder Consistency** Because the work that can be performed in the body of an encoder and decoder is arbitrary, consistency between a class's encoder and decoder is not enforced at compile time. Run-time consistency checks are performed, but the degree to which they are enforced is not defined by the language. A program is informed about encoder and decoder inconsistencies that are detected at run-time as objects are encoded and decoded by the built-in exceptions *RpcEncodeError* and *RpcDecodeError*, respectively.

## 8. Extended-C++ Library

The Extended-C++ library is not part of the language, however it does offer functionality that compliments the language. The library consists of a collection of functions, including the following:

- *lselect* - Similar to the BSD UNIX function *select*; used for multiplexed I/O, but causes only the executing thread to block.

- *lsleep* - Puts a thread to sleep for at least the specified time.
- *lgetpid* - Returns a value that uniquely identifies the executing thread.
- *lpreempt* - Causes the executing thread to be preempted.
- *composeremptr/decomposeremptr* - Encodes/decodes a **remotable** pointer into/from a byte array that can be used to "bootstrap" distributed services such as name servers.

In addition, versions of several functions that cause processes to block waiting for a UNIX signal have been re-implemented to block only the executing thread (e.g., *pause* and *sigpause*)<sup>18</sup>.

## 9. Implementation

Extended-C++ is implemented by a translator that generates C++ code from Extended-C++ code, and a run-time library. The run-time library is implemented in C++ and assembly code. A symbolic debugger (called *edb*) that provides a window-per-thread has also been developed. The language currently runs on Hewlett-Packard Series 300 workstations, which are 680X0-based, and will be ported to Hewlett-Packard RISC-based processors in the near future.

## 10. Conclusions

Extended-C++ has been in use since 1987. The development team has written more than 250,000 lines of Extended-C++ code to implement the many programs that comprise the information management system. These programs currently define 22 remotable classes, 108 transmissible types, and 122 remotable functions.

The generality of Extended-C++ was relatively unhindered by the design constraints placed upon it. The few language definition problems that were encountered were in the area of exception handling, where the treatment of partially constructed or destroyed objects is not as elegant as desired. In addition, it was impossible to support the desired exception handling semantics for inline functions. Therefore, inline functions have the seemingly arbitrary limitation that they can neither raise nor handle exceptions.<sup>19</sup>

Extended-C++ has made the kind of functionality that it provides readily available and useful to a large team of engineers with diverse backgrounds. In the clinical information management system that was developed using Extended-C++, exception handling is the most commonly used Extended-C++ feature. However, concurrency is also extensively used, and a number of processing problems (ranging from multi-threaded user interfaces to serverized databases) have been solved using threads. RPC is the only inter-process communication mechanism used. The garbage collector proved quite valuable as a debugging and memory-profiling tool, but it currently is a goal to explicitly manage memory and to not rely on garbage collection. In addition, substantial amounts of existing software have been successfully integrated *unaltered* with Extended-C++; a notable example of this was the serverization of a standard relational database product.

Perhaps the most significant indicator of the success of Extended-C++ is that almost all discussions about the design and implementation of the clinical information management system used C++ or

18. The entire process will be blocked if the calling function is non-preemptable.

19. These limitations could be avoided if we were willing to modify the C++ translator.



Extended-C++ terminology in lieu of lower-level terminology. This is most exciting because it indicates that the level of abstraction for the development of distributed programs has been raised from that of operating system and machine architecture to programming language. This observation can be taken as a measure of how well the goal of simplifying the development of a distributed object-oriented system was achieved.

## 11. Acknowledgements

The author would like to thank Lance Smith for providing the skill and perseverance that made the Extended-C++ translator a reality, Andy Braunstein, Elaine Calm and Hanfei Yu for the contributions that they made to the Extended-C++ run-time library, and Mike Repucci for providing his programs as the earliest tests of Extended-C++.

## 12. References

- [1] Andrews, G. R., "The distributed programming language SR - Mechanisms, design and implementation," *Software Practice and Experience*, Vol. 12(8): 719-753, Aug. 1982.
- [2] Atkins, M. C., and Nackman, L. R., "The active deallocation of objects in object-oriented systems," *Software Practice and Experience*, Vol. 18(11): 1073-1089, Nov. 1988.
- [3] Barnes, J. G. P., "An overview of Ada," *Software Practice and Experience*, Vol. 10(7): 851-887, July 1980.
- [4] Bennet, J. K., "The design and implementation of distributed Smalltalk," in *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, ACM, pp. 318-330, Oct. 1987.
- [5] Black, A., Hutchinson, N., Jul, E., and Levy, H., "Objects in the Emerald system," in *Proceedings of Object-Oriented Programming, Systems, Languages and Applications*, ACM, pp. 78-86, Oct. 1986.
- [6] Gehani, N. H., and Roome, W. D., "Concurrent C," *Software Practice and Experience*, Vol. 16(9): 821-844, Sept. 1986.
- [7] Goldberg, A., and Robson, D., *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Ma., 1983.
- [8] Grief, I., Seliger, R., and Wheil, W., "Atomic data abstractions in a distributed collaborative editing system," in *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, ACM, pp. 160-172, Jan. 1986.
- [9] Herlihy, M., and Liskov, B., "A value-transmission method for abstract data types," *ACM Transactions on Programming Languages and Systems*, Vol. 4(4): 527-551, Oct. 1982.
- [10] Lampson, B. W., and Redell, D. D., "Experiences with processes and monitors in Mesa," *Communications of the ACM*, Vol. 23(2): 105-117, Feb. 1980.
- [11] Liskov, B., Herlihy, M., and Gilbert, L., "Limitations of remote procedure call and static process structure for distributed computing," Programming Methodology Group Memo 41, MIT Laboratory for Computer Science, Sept., 1984.
- [12] Liskov, B., and Schiefler, R., "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5(3): 381-404, July 1983.

- [13] Liskov, B., and Snyder, A., "Exception handling in CLU," *IEEE Transactions on Software Engineering*, Vol. SE-5(6): 546-558, Nov. 1979.
- [14] Milner, R., Toftes, M., and Harper, R., *The Definition of Standard ML*. MIT Press, Cambridge, Ma., 1990.
- [15] Mitchell, J. G., Maybury, W., and Sweet, R., "Mesa Language Reference Manual," Technical Report C'SL-78-1, Xerox Research Center, Palo Alto, Feb. 1978.
- [16] Nelson, B. J., "Remote procedure call," Technical Report CMU-CS-81-119, Dept. of Computer Science, Carnegie Mellon Univ., Pittsburgh, Pa., May, 1981.
- [17] Panzieri, F., and Shrivastava, S. K., "Rajdoot: a remote procedure call mechanism supporting orphan detection and killing," Technical Report NTU-CL/TRS-200, Newcastle-upon-Tyne University Computing Laboratory, 1985.
- [18] Seliger, R., "Design and implementation of a distributed program for collaborative editing," Technical Report MIT-LCS-TR-350, Laboratory for Computer Science, MIT, Cambridge, Ma., Jan. 1986.
- [19] Shaw, Robert A., "The empirical analysis of a Lisp system," Tech. Rep. CSL-TR-88-351, Computer Systems Laboratory, Stanford University, Palo Alto, Ca., Feb. 1988.
- [20] Tanenbaum, Andrew S., "A tutorial on Algol 68," *ACM Computing Surveys*, Vol. 8(2): 155-190, June 1976.

## Appendix - An Example of a Distributed Extended-C++ Program

The following example illustrates the components and contents of a distributed Extended-C++ program. The program is implemented as a pair of programs called the *client* and the *server*. In order to initially make contact with each other, a globally accessible name server (not shown in the example) is used. The name server allows the *server* to register the value of a **remotable** pointer with a description of the **remotable** object pointed to. The *client* can then ask the name server to look up a particular description and return the corresponding **remotable** pointer. In the example, the name server is implemented by the class *NameServer*.

The source files common to both the *client* and *server* programs are:

```
//----- Service.h -----

class Service
{
public:
    Service();
    void doSomething() remotable;
};
```

The source files for the *client* program are:

```
//----- ServiceC.c -----
```

```
#include "Service.h"
```

```
remotable class Service = 0;
```

```
//----- MainC.c -----
```

```
#include "NameServer.h"
```

```
#include "Service.h"
```

```
extern "C"
```

```
{
```

```
    void exit(int);
```

```
}
```

```
NameServer ns;
```

```
main()
```

```
{
```

```
    remotable Service *sp;
```

```
    while (1) {
```

```
        // Keep looking (every second) for the service until find it.
```

```
        // When do find it, cast untyped remotable pointer.
```

```
        sp = (remotable Service *)ns.find("Service");
```

```
        lsleep(1000);
```

```
        break;
```

```
    }
```

```
    except when NotFound;;
```

```
    end;
```

```
    sp->doSomething();
```

```
    exit(0);
```

```
}
```

The source files for the *server* program are:

```
//----- ServiceS.c -----

#include "Service.h"

remotable class Service = 0;

Service::Service()
{
    // body of constructor
}

void Service::doSomething() remotable
{
    // function body
}

//----- mainS.c -----

#include "NameServer.h"
#include "Service.h"

Service s;
NameServer ns;

main()
{
    ns.add("Service", (@s);
}
```

The *server* creates the static objects *ns* and *s*. A description for *s* and a **remotable** pointer to *s* are registered with *NS*. The thread that executes the *server* function *main* then terminates, but the *server* process does not terminate; instead it sleeps waiting for **remotable** function calls.

The *client* creates the static object *NameServer NS* and finds in *NS* the name of the service "Service" and a **remotable** pointer to an object that implements the service (i.e., the object *S* in the *server*.) The *client* then executes the **remotable** function *Service::doSomething* using the **remotable** pointer.

When the *server* receives the **remotable** function request message, a new thread is implicitly created to execute the **remotable** function call *Service::doSomething* on the object *S*. When the call returns, *client* exits.

The modules required to link the *client* are:

```
mainC.o  ServiceC.o  NameServer.o
```

and the modules required to link the *server* are:

```
mainS.o  ServiceS.o  NameServer.o
```

Note that the *client* only needs to link with the module that instantiates the **remotable** class *Service*; it does not need the module that implements this class. This is because the *client* uses the class *Service* as a truly remote class.



# The C++ Information Abstractor

Judith E. Grass

Yih-Farn Chen

AT&T Bell Laboratories

Murray Hill, NJ 07974

The *C++ Information Abstractor*, *cia++*, builds a database of information extracted from C++ programs. The database can serve as a foundation for the rapid development of C++ programming tools. Such tools include tools that graphically display various views of the program structure, tools that answer queries about program symbols and relationships, and tools that extract self-contained components from a large system. *Cia++* is a new abstractor implementation based on the design of the *C Information Abstractor*. This paper describes the conceptual model of *cia++* and examples of relational views and applications developed on the C++ program database. It also presents some aspects of the implementation.

## 1 Motivation

C++ is rapidly becoming a language of choice for developing large application programs due to its object-oriented paradigm. Many new C++ programmers come from the C programming community. The transition from C to C++ is made easier by the fact that these languages share a common core. It is made more difficult by the lack of tools to support C++ program development. C programmers enjoy the support of many program development and analysis tools. Unfortunately, few of these tools transfer gracefully to C++. The lack of tools designed for C++ discourages developers from adopting it. The critical need for C++ programming tools is only beginning to be seriously addressed.

This paper describes the *C++ Information Abstractor* (*cia++*) and the system of tools built around the database that it generates<sup>1</sup>. *Cia++* extracts information about the various entities<sup>2</sup> declared in a C++ program and the relationships between those entities. The abstracted information is stored in a database that can be used by any number of tools. Each tool provides a different view of the way complex C++ programs are structured.

The *C Information Abstractor* [1, 2], usually referred to as *cia*, builds a program database for C. *Cia++* provides the functionality of *cia* for C++ programs. Because C++ is a semantically richer language than C, *cia++* must reflect new entity attributes and relationships between entities that do not exist in C programs. One example of such a relationship is inheritance, a relationship between two classes. This paper presents a specification of the conceptual model and database schema used to store the information abstracted from a C++ program. Some aspects of a *cia++* implementation are also explored here.

## 2 The Design Rationale

*Cia++* is designed to be an extensible platform for rapidly developing program analysis tools. Several features of *cia++* facilitate this:

<sup>1</sup>By convention, *cia++* written in lower case letters refers just to the program that abstracts data from a C++ program. *CIA++* written in capital letters refers to the system made of that abstractor and all of the tools that use the database it generates.

<sup>2</sup>Entities include files, macros, types, functions and variables.

- *Separation of Information Extraction and Presentation:* The process of extracting information from a program and the process of presenting that information should be separate. This eliminates the need for each C++ analysis tool to duplicate the parsing process and allows C++ tools to share the information and present it in different ways. A similar doctrine was developed in the *Interlisp* [3] project. Unfortunately, many C and C++ tools today still violate this basic principle.
- *A Well-Conceived Conceptual Model:* A conceptual model based on the entity-relationship model [4] was designed to describe the entities, attributes, and relationships in C++ programs. The model serves as an accurate high level specification for the contents of the program database.
- *Relational Database:* To support reuse, *cia++* stores abstracted information in the form of a relational database. This makes it possible for that information to be accessed by a wide variety of existing database query systems.
- *Incremental Database Construction:* *Cia++* builds and maintains a program database for large systems efficiently because it allows incremental updates of a database. When a set of C++ source files is updated, only that portion has to be reabstracted to update the database.

These basic design principles give tools based on *cia++* an advantage over many traditional development tools, which usually merge the information extraction and presentation processes in a single tool and do not share the information with other tools.

The C program database generated by *cia* has had many new C tools developed on top of it. These tools were developed in a matter of days by using the common database, but different approaches. Some tools were written as *awk* or *ksh* scripts that call database query commands. Others are C programs that call a query library. Other applications have been written that use INGRES [5] or DataShare<sup>3</sup> queries to extract information from the C program database.

With minor changes to adjust to the different format of the C++ program database, these tools can be applied to C++ programs. Many new tools can be developed in a short period of time by sharing the well structured information in the database. This makes it possible to offer C++ programmers a higher level of tools support than has been possible up to now.

Our conceptual model for a C++ program database contains the information to answer a wide variety of queries, including

- what is the inheritance hierarchy of this class?
- what are all the public, inherited members of this class?
- where are all the references to this class member?
- where are all the declarations of this overloaded function?

It can be time consuming and difficult to discover the answers to these questions if no more support is available than huge collections of source files, *grep* and a text editor.

A program database can be used as a foundation to build several types of browsers as well. This has been done with the C program database. The C++ program database should be a good foundation for building browsers as well; however, we would like to stress that browsers are only one of many applications of the database.

<sup>3</sup>DataShare is an AT&T proprietary database manager written by Rick Greer.

### 3 The Conceptual Model

A well defined conceptual model is a key ingredient in the success of an information abstraction system. The conceptual model specifies which software entities, relationships and attributes should be stored in the program database. It also serves as a requirements specification for the C++ information abstractor. The current version of *cia++* tracks only *global* entities, members of *global* types and interactions between them. This succinct model limits the size of the resulting database, yet provides sufficient information for performing a large number of browsing and analysis tasks.

#### 3.1 Entities

The *CIA++* conceptual model recognizes five kinds of software entities in a C++ program. These are *files*, *macros*, *types*, *variables* and *functions*. All of the entities present in C++ programs can be fit into this framework. For example, definitions of overloaded operators are considered to be function definitions and data members of a class are treated as variables.

Appendix A contains an abridged C++ program with examples of the following entities [6] :

<i>file:</i>	stream.h, tree.h, tree.c, main.c
<i>type:</i>	Tree, Node, IntNode, UnaryNode, BinaryNode, ...
<i>variable:</i>	Tree::p, UnaryNode::op, UnaryNode::opnd, BinaryNode::left, ...
<i>function:</i>	operator<<(ostream&, const Tree&), Node::print(ostream&), ...

Each one of the five kinds of *cia++* entities has a unique list of attributes, which specify various information about an entity declaration. *Cia++* records all of the declarations (including definitions) that are explicit in the source code. In addition, it records definitions of functions and variables generated by *cfront*. These implicit definitions usually carry unusual names and are easy to identify.

#### 3.2 Relationships

*Cia++* records reference relationships between global entities and members of classes, structures and unions. It also tracks references to static variables and functions declared global to a file and extracts information about the relationships between the recorded entities. Table 1 lists all of the reference relationships recorded by *cia++*. Given two global entities or members A and B, there is a relationship between them if:

- A is a file and B is another file included by A. This is an *include* relationship from A to B.
- Both B and A are types and B inherits A. This is an *inheritance* relationship from B to A.
- B is a type and A is a function or type, and B nominates A as a friend. This is a *friendship* relationship from B to A.
- Neither A nor B is a file, B is neither a friend nor a parent class of A, and some declaration of A refers to B. This is a *reference* relationship from A to B.

Each relationship has a *usage* attribute that stores information about where relationships were discovered in the source files. The information is mainly to be used by C++ browsers.

The *include* and *reference* relationships also exist in the C program database; however the inheritance and friendship relationships are unique to C++ programs and they have additional attributes.

The small sample program contains many reference relationships. Here are a few examples:

Entity Reference Relationships		
entity A	entity B	definition
function	function	function A refers to function B
function	variable	function refers to variable
function	type	function refers to type
function	macro	function refers to macro
variable	function	variable refers to function
variable	variable	variable A refers to variable B
variable	type	variable refers to type
variable	macro	variable refers to macro
type	variable	type refers to variable
type	type	type A refers to type B
type	macro	type refers to macro

Table 1: Table of Reference Relationships

entity 1	relationship	entity 2	comment
tree.h	includes	<stream.h>	file to file
Node	refers to	Tree	type to type
Tree	is a friend of	Node	type to type
IntNode	inherits	Node	type to type
BinaryNode::left	refers to	Tree	mem_variable to type
Tree::~Tree	refers to	Node::use	mem_function to mem_variable
Tree::Tree	refers to	IntNode::IntNode	mem_function to mem_function
main	refers to	Tree	function to type
IntNode::print	refers to	IntNode::n	mem_function to mem_variable

## 4 Relational Views

The program database generated by *cia++* contains a lot of information packed into a relatively small space. It uses an ASCII representation that is human readable, but the amount of information and the terseness of its representation make direct use untenable without relational database tools.

When *cia* was written, a specially tailored database retrieval system called *InfoView* was also developed to extract information from the C program database. Most *InfoView* commands can also be used with the database generated by *cia++*, but because *InfoView* was specifically designed to work with *cia*, it cannot present information about the additional fields and relationships tracked by *cia++*. As a result, it cannot easily answer the queries presented in Section 2.

There are great advantages to adopting a more general database manager for *CIA++* that could be used with a broader family of information abstractors. *DataShare* appears to be well suited for this purpose<sup>4</sup>. *DataShare* provides a powerful query language for advanced data processing and a translation system that converts queries into efficient C programs. Several *InfoView*-like commands have been developed in a short period of time using *DataShare*.

The following sections show two major types of queries that can be run on the *CIA++* database: entity queries and relationship queries. In each of these examples, the first line shows a computer prompt (\$) and a database query command. The lines that follow are the results of that query.

<sup>4</sup>Several other database management systems have been used with *CIA* and should be useful with *CIA++* as well.



## 4.1 Entity Queries

Entity queries deal mainly with attributes of all entities. They take two arguments, entity kind and entity name, and return the attribute information. The first sample query displays all of the declarations of a function called `print`. This finds all functions with that name regardless of overloading or class membership.

```
$ Def function print
file      dtype      ms spec name      bline df
=====
tree.h    void ( )(ostream&)  pv vi  BinaryNode::print  53  df
tree.h    void ( )(ostream&)  pv vi  IntNode::print     32  df
tree.h    void ( )(ostream&)  pt vi  Node::print        10  df
tree.h    void ( )(ostream&)  pv vi  UnaryNode::print   40  df
```

The output shows that there are three private (pv) and one protected (pt) member function definitions (df) of `print`. These functions are all declared virtual (v) and inline (i) in `tree.h` and they all have the same signatures.

An option `-u` is provided to present the output information in a raw form. For example, the following query shows all the attributes of the member function `UnaryNode::print` in its unformatted form, which makes pipeline processing convenient.

```
$ Def -u function UnaryNode::print
578:print:p:tree.h:void ( )(ostream&):40:44:df:pv:UnaryNode:vi
```

This example also shows that an entity name can be qualified with its parent class name<sup>5</sup>. Such a qualified name takes this form: `class_name::name`, where `name` is a member of `class_name`.

It was easy to implement a command, `Viewdef`, that shows the definitions and declarations of selected entities as they appear in the source code. `Viewdef` simply runs an `Awk` script on the output of `Def -u` commands. For example, here it retrieves the definition of `UnaryNode::print`:

```
$ Viewdef -f -n fu UnaryNode::print
tree.h:40      void print (ostream& o) {
tree.h:41          o << "(" ;
tree.h:42          o << op ;
tree.h:43          o << opnd ;
tree.h:44          o << ")"; }
```

The option `-f` turns on the printing of filenames and `-n` turns on the printing of line numbers.

*Selection clauses* in the form of `attribute=value` can be attached at the end of the `Def` or `Viewdef` command line to restrict the query output further. *Egrep* style regular expressions also can be used to specify entity or attribute names. For example, this query displays all private member functions of the class `BinaryNode`:

```
$ Def function BinaryNode:: mscope=pv
file      dtype      ms spec name      bline df
=====
tree.h    void ( )(char *, Tree, pv i  BinaryNode::__ct  52  df
tree.h    void ( )(ostream&)  pv vi  BinaryNode::print  53  df
```

<sup>5</sup>It is also possible to further restrict the query to functions that take a specific argument list. This requires a bit more command syntax than it is appropriate to discuss here.

If the name component of a qualified name is missing, then it's equivalent to a `.*`, so it matches any member in the specified class.

## 4.2 Relationship Queries

Relationship queries deal mainly with the reference database. Each reference query takes four arguments that specify the parent entity kind, parent entity name, child entity kind, and child entity name.

If a query is intended to find all entities that refer to a known entity, then the child entity name is usually specified as a wildcard, `"-"`. For example, this query shows all of the types that refer to the class `Node`. The results of the query are displayed in the formatted style.

```
$ Ref type - type Node
k1 file1      name1      k2 file2      name2      rk
== =====
t tree.h      Tree        t tree.h      Node        r
t tree.h      Tree        t tree.h      Node        f
t tree.h      IntNode     t tree.h      Node        i
t tree.h      UnaryNode   t tree.h      Node        i
t tree.h      BinaryNode  t tree.h      Node        i
```

This shows that the classes `IntNode`, `UnaryNode` and `BinaryNode` are all derived from `Node`. The class `Tree` is a friend and has another simple reference relationship to `Node`. The `Ref` command also takes selection clauses. The command `"Ref type - type Node rkkind=i"` would return only the last three rows of information.

On the other hand, if a query is intended for finding all entities that are referred to by a known entity, then the parent entity name is usually specified as a wildcard. For example, this query returns all the variables referred to by `main`. There happens to be only one.

```
$ Ref function main variable -
k1 file1      name1      k2 file2      name2      rk
== =====
p main.c      main        v CC/iostream.h cout      r
```

The output of `Ref` can also be displayed in an unformatted style.

## 5 Applications

The strength of the `CIA++` system lies in the tools that can be built on top of the database. This section describes some of the experimental tools we are currently using with the `CIA++` database.

Queries presented in section 4 are most often used to answer specific questions about a small number of program components. The database can also be used to investigate relationships between large numbers of components. Frequently C++ programmers want to have a global picture about how the pieces of the program are interconnected. This kind of information can be presented in a number of ways, but often the most easily understood representation is graphical.

Several tools have been developed to extract structural information from the C++ program database and display the structure as a graph. The following sample graphs were generated from the database for the program in Appendix A.

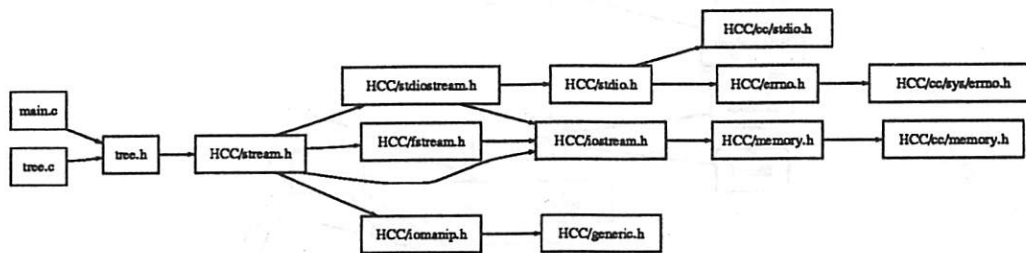


Figure 1: A File Include Graph

## 5.1 A File Include Graph

Often it can be difficult to know exactly what files get pulled into a C++ module when it is compiled. Part of that is because files that are explicitly included in the source file may include other files, which in turn may bring in more files, and so on. This is also affected by the environment. File names that are not completely specified may be resolved in several ways. A graph that shows the inclusion relationship can answer many such questions. Figure 1 shows the file include graph for `tree.c` and `main.c`. It was generated by the following command:

```
$ Dagen file file | sed -e 's,/usr/local/include/CC,HCC,' | dag -Tps | lpr
```

*Dagen* takes two arguments that specify the relationships to be drawn and outputs a graph description that can be processed by *dag* [7]. The *sed* command is used to abbreviate long pathnames. This example also shows that users can easily insert their own filters to alter the output.

*Dagen* accepts all kinds of relationships in the C++ program database. For example, “*Dagen function function*” gives a function call graph. The following three sections show additional examples related to the type to type relationships.

## 5.2 A Type Inheritance Graph

The semantics of C++ introduces many concepts related to classes. The inheritance relationship is central to object-oriented programming. Often it is useful to represent class hierarchies defined by inheritance as graphs. Figure 2 shows the type inheritance hierarchies for all the classes used in `tree.c`. It was generated by the following command:

```
$ Dagen type type rkind=i | dag -Tps | lpr
```

Similar to *Def* and *Ref*, optional selection clauses can be used to limit the output of *Dagen* to a subset.

`Tree.c` defines only one class hierarchy directly. That is the very simple class hierarchy rooted at the `Node` class. The other two hierarchies shown here are from the *iostream* library [8] pulled in by including *stream.h*. The hierarchy rooted at the class `ios` is interesting. Single inheritance class hierarchies always should appear as trees. *Ios* is the root of a hierarchy that uses multiple inheritance. This cannot be represented with a simple tree.

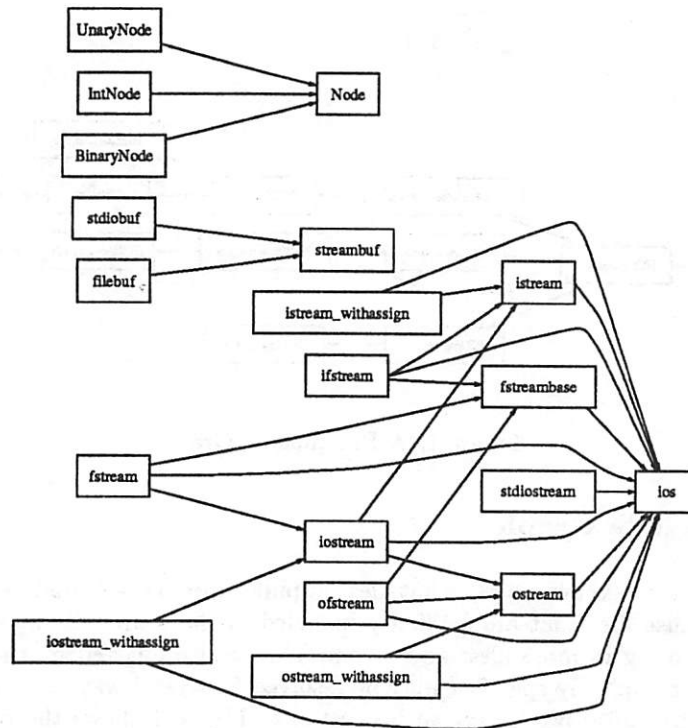


Figure 2: A Type Inheritance Graph

### 5.3 A Friendship Graph

Friendship is another relationship between classes and other entities that may be graphed. Figure 3 shows the friend relationships found in *tree.c*. There is a cycle in the graph because *Node* is declared as a friend of *Tree* and *Tree* is declared as a friend of *Node*. The figure was generated by the following command:

```
$ Dagen type type rkind=f | dag -Tps | lpr
```

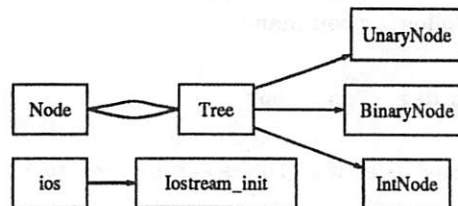


Figure 3: Friend Relationships



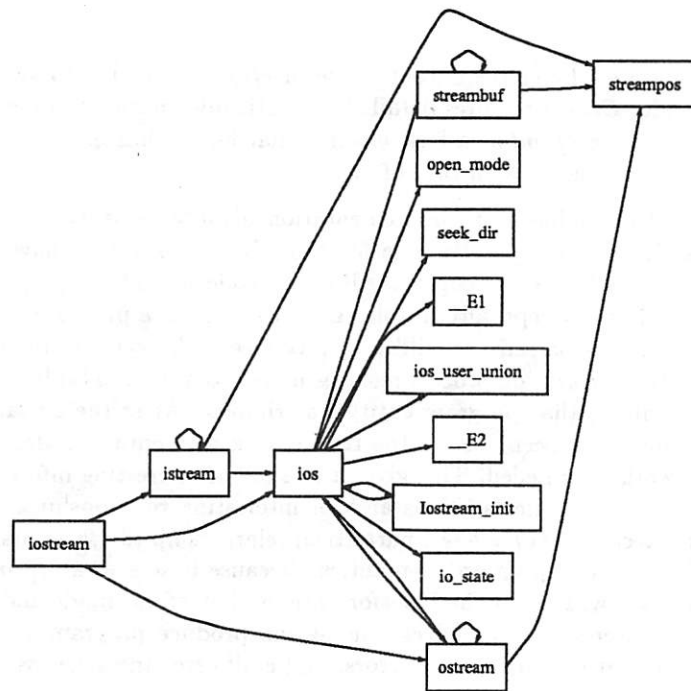


Figure 4: The Subsystem of iostream

## 5.4 Subsystem Extraction

Graphical tools can also be used to isolate self-contained components of a program. Type to type relationships define one kind of interesting subsystem. Figure 4 shows the subsystem of `iostream`; i.e., all the data types referred to directly or indirectly by `iostream`. This includes types that `iostream` uses for its members, the types that are friends and the types it inherits. The types `__E1` and `__E2` are names that `cfront` generated for the two anonymous enumerator types defined inside the class `ios`. The graph does not include relations to such basic types as `int`, `char` or `long`. The figure was generated by the following command:

```
$ Subsys -e type iostream type | Dagen -i type type | dag -Tps | lpr
```

## 6 Implementation Notes

The design and implementation of `cia++` involve developing a database schema and writing code to populate a conforming database with information extracted from a C++ program. In designing a database schema and format, we tried to deviate as little as possible from the schema developed earlier for C. The commonality of structure makes it possible to directly use some `cia` based tools and to rapidly adapt others. It also makes it possible for C programmers who have used `cia` to more easily adapt to C++ and `cia++`. More importantly, it allows us to merge C and C++ program databases with little effort, which is critical for projects that involve both C and C++ programs.

`Cia++` was implemented using `cfront` version 2.0 as a starting point. This approach has many advantages and avoids many potential pitfalls of writing an abstractor from scratch. Some of the advantages derive from the fact that C++ is a young language and undergoing change. This was especially true while the abstractor was being built. At the time, C++ version 2.0 was not officially

released. Developing *cia++* by embedding its code in *cfront* allowed it to keep in synchrony with the changes being made. *Cia++* requires detailed semantic information to extract relevant relations and entity attributes. Using *cfront* as a base ensures that *cia++* has an accurate understanding of the semantics that agrees exactly with that of *cfront*.

Using *cfront* as a platform has many implementation advantages as well. No parsing of any kind need be written, as *cia++* uses *cfront*'s code for this. Nor does *cia++* have to deal with any of the intricacies of name resolution or scoping. All of the code for this already exists in *cfront*. The way the abstractor works is conceptually simple. *Cfront* processes a program one external definition at a time. The definition is parsed to build a syntax tree. The syntax tree then goes through a declaration process. The declaration process resolves names to a symbol table reference that contains complete information about that program entity's attributes. After the declaration process, some program transformations may occur before the tree is walked to emit C code. *Cia++* allows *cfront* to do its declaration work unimpeded. This gives a tree full of interesting information. *Cia++* walks this tree looking for new entity declarations and for interesting relationships. Anything it finds, it adds to its database. Because *cia++* has a parasitical relationship to *cfront*, its view of a program's semantics is *cfront*'s view of a program's semantics. Because it sees what *cfront* sees, it can report on the code that the user wrote and the transformations that *cfront* made and know the difference between them. As an interesting side effect, *cia++* can produce program databases for programs that are incomplete or that contain syntax errors, and emit error messages as well.

This description makes the process of reusing *cfront* code seem much easier than it really is. The source code for *cfront* is very complex and not at all easy to understand or modify. Sometimes finding the exact spot in the code where all the information *cia++* needs is present and where nothing needed has been destroyed is literally impossible. So far, experimentation has usually shown a way to get around such dilemmas. In general, we have avoided making deep changes in the code of *cfront*.

## 7 Conclusion and Status

The current version of *cia++* has been successfully run on complex C++ programs, including the *iostream* package, *cfront*, and *cia++* itself. Our current vision of *cia++* should fill the need for a wide range of static program analysis tools. Eventually this will develop into a base for browser-like tools. Some components of *cia++* could probably be incorporated into a source language debugger. Our immediate goals are more limited.

We would like to thank Rick Greer for his help in showing us how to write efficient DataShare queries. Peter Selfridge and Peter Kirsulis reviewed a draft of this paper and provided valuable comments.

## References

- [1] Y. F. Chen. The C Program Database and Its Applications. In *Usenix Summer 1989 Conference Proceedings*, Baltimore, MD, June 1989.
- [2] Y. F. Chen, Michael Nishimoto, and C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, March 1990.
- [3] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. *IEEE Computer*, 14(4):25-34, April 1981.
- [4] P. S. Chen. *The Entity-Relationship Approach to Software Engineering*. Elsevier Science, 1983.
- [5] M. Stonebraker, E. Wong, R. Kreps, and G. Held. The Design and Implementation of INGRES. *ACM Transactions on Database Systems*, 1(3):189-222, September 1976.
- [6] Andrew Koenig. An Example of Dynamic Binding in C++. *Journal of Object-Oriented Programming*, 1(3), August 1988.
- [7] E. R. Gansner, S. C. North, and K. P. Vo. DAG - A Program that Draws Directed Graphs. *Software - Practice and Experience*, 18(11), November 1988.
- [8] AT&T. UNIX System V AT&T C++ Language System: Release 2.0, Library Manual, 1989. Select code 307-145.

## Appendices

### A A Sample C++ Program

Reprinted with permission from the *Journal of Object Oriented Programming*. This appeared in Volume 1, Number 3, August/September 1988.

```
/*<tree.h>*/

1  #include <stream.h>
2
3  class Node {
4      friend class Tree;
5      friend ostream& operator<< (ostream&, const Tree&);
6  private:
7      int use;
8  protected:
9      Node() { use = 1; }
10     virtual void print (ostream&) { }
11     virtual ~Node() { }
12 };
13
14 class Tree {
15 public:
16     Tree(int);
17     Tree(char*,Tree);
18     Tree(char*,Tree,Tree);
19     Tree(const Tree& t){ p = t.p; ++p->use; }
20     ~Tree() { if (--p->use == 0) delete p; }
21     void operator=(const Tree& t);
22 private:
23     friend class Node;
24     friend ostream& operator<< (ostream&, const Tree&);
25     Node* p;
26 };
27
28 class IntNode: public Node {
29     friend class Tree;
30     int n;
31     IntNode (int k): n (k) { }
32     void print (ostream& o) { o << n; }
33 };
34
35 class UnaryNode: public Node {
36     friend class Tree;
37     char *op;
38     Tree opnd;
39     UnaryNode (char* a, Tree b): op (a), opnd (b) { }
40     void print (ostream& o) {
41         o << "(" ;
42         o << op ;
```

```

43         o << opnd;
44         o << ")"; }
45     };
46
47     class BinaryNode: public Node {
48     friend class Tree;
49     char* op;
50     Tree left;
51     Tree right;
52     BinaryNode (char* a, Tree b, Tree c): op (a), left(b), right(c) { }
53     void print (ostream& o) { o << "(" << left << op << right << ")"; }
54 };

```

/\*<tree.c>\*/

```

1  #include "tree.h"
2  Tree::Tree(int n) { p = new IntNode (n); }
3  Tree::Tree(char* op, Tree t) { p = new UnaryNode (op, t); }
4  Tree::Tree(char* op, Tree left, Tree right) { ... }
5  void Tree::operator=(const Tree& t){ ... }
6  ostream& operator<< (ostream& o, const Tree& t) { ... };

```

/\*<main.c>\*/

```

1  #include "tree.h"
2  main()
3  {
4      Tree t = Tree ("*", Tree("-", 5), Tree("+", 3, 4));
5      cout << t << "\n";
6      t = Tree ("*", t, t);
7      cout << t << "\n";
8  }

```





# Adding New Code to a Running C++ Program

Sean M. Dorward, Ravi Sethi, Jonathan E. Shopiro

AT&T Bell Laboratories<sup>1</sup>

Dynamic linking of new code adds considerable flexibility to a programming system, but usually at the expense of type-safety at the interface between old and new code. Inheritance in C++ supports a type-safe special case of dynamic linking, where the new code is that of a class derived from an already existing base class. Once a new derived class is linked into a running program, using an incremental linker, the original program can safely manipulate derived objects in addition to base objects, since an object of a derived class (a subtype) can appear where an object of a base class (a supertype) is expected.

But, how might the existing code create objects of a class that is yet to be defined? Our solution is to maintain a map from class names to their object-creation functions. Class names are strings that can be passed dynamically to the existing code. Technically, a class name is mapped to a so-called *class object* for the class; the class object supports object creation. In our implementation, the code for maintaining maps and class objects is inserted by a preprocessor. This solution stays within the C++ language, so it is portable across C++ compilers.

Finally, the implementation of virtual functions can be exploited to change the class of an object on the fly; we explain under what conditions this transformation is type-safe.

## 1. Introduction

Let *dynamic linking* refer to the ability to add new code to a running program — new code that can then be accessed from within the old code. By contrast, let *incremental linking* refer to the ability to add code that the running program already knows about.

This paper describes a type-safe, efficient, and portable implementation of dynamic linking in C++. The implementation allows a running program to incorporate a new class, derived from a class that the program already knows about; the old code is then able to manipulate and create objects of the new class. Derived classes and inheritance in C++ are reviewed in Section 2. The interface between the old and the new code is considered informally in Section 3, in preparation for an extended example in Section 4. Further details of the interface appear in Section 5, along with a description of a preprocessor that inserts appropriate code to dynamically manage the interface.

<sup>1</sup> Present addresses: Sean M. Dorward, 511 Laughlin Hall, Princeton University, Princeton, New Jersey 08544; Ravi Sethi, AT&T Bell Laboratories, Murray Hill, New Jersey 07974; Jonathan E. Shopiro, AT&T Bell Laboratories, 184 Liberty Corner Road, Warren, New Jersey 07060.

## Motivating Applications

A server such as a network controller or an operating system is expected to provide continuous service, despite the need for occasional updates to the server. If running programs can be updated on the fly, then service need not be disrupted. The alternative is to stop the server and load a new version of its software.

This paper is motivated by Fraser's [1979] erector-set approach of assembling a data network from plug-to-plug compatible modules. "Initially one could start with a small catalogue [of modules] and add to it as the market for new services expands and technology makes improvements possible." If the software could mirror the hardware, a new module might be added to a running network by plugging in its hardware and incorporating its code into the software for the network. Chesson [1979] states this goal as one of providing "an extensible parts list of software modules that can be used singly or in combinations in order to satisfy networking requirements."

Another example, from Stroustrup [1987], is a CAD/CAM system that allows new components to be defined and added to a layout currently on the screen. "Using traditional compile, load, and go techniques it is difficult to add the new component class to the layout program without having to re-create the 'current layout' from scratch after re-compilation of the layout program."

## Related Work

An outline of the proposal in this paper was reported by Stroustrup [1987]. The proposal has since been refined as implementation details were worked out.

On the surface, the goals of this paper are similar to those of CLAM, a system for building graphical user interfaces described by Call, Cohrs, and Miller [1987], who "would like an extensible environment that would allow clients to dynamically create objects, layers, and extensions as needed." The distinction is that CLAM relies only on incremental linking; at run time, it allows existing code to request code that it already knows about.

Incremental linking of C and C++ programs is well established; see for example Puttress and Goguen [1986], and Quong [1989].

Programming environments for languages such as Lisp (Teitelman and Masinter [1981]) and Smalltalk (Goldberg [1984]) have long supported incremental updates.

## 2. Derived Classes and Inheritance in C++

The C++ inheritance mechanism allows formal parameter and operand types to be viewed as constraints on the types of the actual arguments. Such constraints are automatically met by derived objects. A definitive account of inheritance in C++ appears in the reference manual by Ellis and Stroustrup [1990]; this section contains a brief review.

### Pass a Reference or Pointer to a Derived Object

Central to inheritance in C++ is the notion of a class declared as an extension of an existing class. The extension is known as a *derived* class and the existing class is known as a *base* class. The derived class implicitly contains — *inherits* — all the members of the base class. The usual syntax of class derivation is<sup>2</sup>

<sup>2</sup> The keyword `public` ensures that the derived class preserves the visibility of all members of the potentially multiple base classes. The derived class is therefore a subtype of its base classes. For more on subtypes, see Snyder [1986] or Sethi [1989].

```
class <derived> : public <base> {
    <declarations for members added by the derived class>
};
```

A function call into dynamically linked code is *prima facie* type-unsafe, since the new code may not even have existed when the calling code was written and compiled; hence, the call could not possibly have been checked with respect to the function arguments and return type. The C++ inheritance mechanism provides a solution to this problem because the declaration of a base class can be viewed as an interface specification that applies to all derived classes. Thus objects of a new derived class, even a derived class that does not exist when the original code is written, are manipulated in a type-safe way by treating them as if they were objects of the base class.

For example, consider classes T and E declared by

```
class T {
    ...
    String text;
    virtual void print();           // print the text string
};
class E : public T {
    ...
    int permissions;
    void exec();                   // execute the text string
    void print();                   // check permissions and print
};
```

Here, derived class E inherits the member `text` from the base class T, and redefines the function `print`. The role of the keyword `virtual` will be reviewed in a moment.

A reference to an object `e` of derived class E can be passed as a parameter to a function that expects a reference to a base object. An example is a function `f` declared by

```
void f(T& x) { ...; x.print(); ... }
```

The function call `f(e)` is legal because `e` implicitly contains all the members of class T that are visible to the code for `f`. Alternatively, `e` matches the constraints imposed on the formal parameter of `f`.

### Call New Code Through Virtual Functions

Function names declared to be virtual in a base class are type-checked statically, but they are bound dynamically. Static checking works as follows: the type of a virtual function serves as a constraint on any redefined versions of the function in derived classes; the constraint is checked when a derived class is compiled. Dynamic binding works as follows: if an object belongs to a derived class, then the version of the virtual function in that class is used.

In the preceding code fragment, the body of function `f` contains a call `x.print()`. Since it is virtual in the base class T, the function name `print` denotes the function belonging to the actual parameter denoted by `x`. Actual parameters are available only at run time, and can change from call to call, so the actual function denoted by `print` is determined at run time. (Without the keyword `virtual` in the base class, the version of `print` in the base class would be used even if `x` denotes a derived object.)

Dynamic binding of virtual functions occurs when objects are passed by reference or via explicit pointers, but not when objects are passed by value, the usual parameter-passing method in C++. For example, the function *g* declared by

```
void g(T x) { ...; x.print(); ... }
```

receives its argument by value, while the previous function *f* receives its argument by reference. If *g* is called with a derived object, as in *g(e)*, the formal parameter *x* will be a local variable of the base type *T*. The version of *print()* called from within *g* will then be that belonging to the base class.<sup>3</sup> Under call-by-value *x* is initialized by creating a copy of the base part of the argument, instead of a reference to the derived argument as in the previous example.

### 3. Creating New Objects from Within the Old Code

The direct way of creating an object on the free store at run time is to apply the *new* operator to its class name. This direct way requires the class name to appear in the source text, so it can be used only for known classes. Below, the *new* operator will be applied indirectly to avoid mention of the class name at the point where an object is created.

Objects of a class *Base* can be created by evaluating the expression

```
new Base (<parameters for initialization>)
```

The result is a pointer to the created object. For simplicity, initialization parameters will be omitted — in which case, the parentheses around the parameters can be omitted as well. The simpler expression is *new Base*. The following program fragment provides a context for this expression:

```
Base* p;           declare p to be a pointer to an object of class Base
...
p = new Base;      leave p pointing to a fresh object
```

Now, suppose that class *Derived* is derived from *Base*. The following three-part approach to the indirect construction of *Derived* objects is a simplification of the approach taken later in this paper:

1. *Class object*. Associated with both the base and derived classes are special objects, called *class objects*, generated by our preprocessor.<sup>4</sup> Class-object names have the suffix *Object* attached to the class name.
2. *Object-creation function*. One of the members of *DerivedObject* is a function *mk* that, in effect, uses *new* to create objects:

```
Derived *mk() { the effect is to return new Derived; }
```

Hence, the result of evaluating the following expression is a pointer to a fresh *Derived* object:

```
DerivedObject.mk()
```

3. *Map from class names to their class objects*. Class objects are accessed indirectly, through an associative array or map from class names (strings) to class objects. Maps belong to class objects, and are maintained by code inserted by the preprocessor. The map within

<sup>3</sup> Some versions of the C++ compiler incorrectly fail to distinguish between *f* and *g*.

<sup>4</sup> The term class object is motivated by the notion of classes (which are objects) in Smalltalk (Goldberg and Robson [1983]).



BaseObject provides access to classes yet-to-be-derived from class Base. The syntax for table lookup is borrowed from array indexing, by overloading the [] operator. Thus,

```
BaseObject["Derived"]
```

points to the class object for class Derived. A Derived object can therefore be created by evaluating

```
BaseObject["Derived"]->mk()
```

Maps are updated automatically when a new class is linked into a running program. The initialization code for the new derived class object adds an entry in the map for the base class. For example, an entry mapping Derived to DerivedObject is added to the map within BaseObject. (For technical reasons, we did not use the maps available through the library Map.h, due to Koenig [1988].)

Further details appear in Section 5.

#### 4. Motivating Example: A Toy File System

The main example of this paper is a toy file system loosely modeled on the UNIX® system. Throughout this section, "file" refers not to a UNIX system file, but rather to an object in the example system. The toy file system begins as a "flat" collection of text files, and is dynamically updated to permit read-write-execute permissions on text files and hierarchical directories. This section sketches an implementation of the file system.

##### The Flat File System

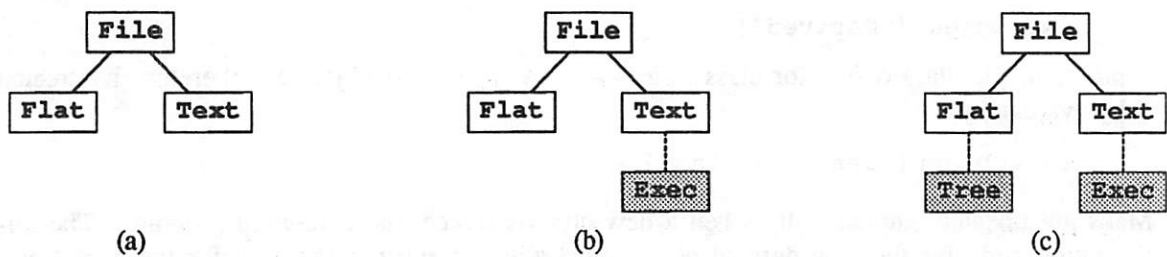
At initiation, the system consists of a base class File, with derived classes Flat and Text; see Figure 1. File and its derived classes have a virtual function, parse, which interprets command lines. A command results in a call to a member function of the same name. The commands supported by the initial classes include those in Figure 2. Commands that are not recognized by the parse function in a derived class are passed to the parse function in the base class.

Interaction with the file system is through a trivial reader object. The reader keeps track of a file as the *current* file, and prompts with its class and file name; each class has a virtual function kind that returns the class name. At start-up, the system consisting of a directory root of derived class Flat, so the initial prompt is

```
Flat!root:
```

Lines typed to the reader are simply passed to the parse function of the current file for interpretation and response.

The next few commands demonstrate some operations on text files. Class Flat supports a command for creating read-write files of class Text. The commands create a text file foo, change the current file to foo, append some text to it, and then change the current file back to root:



**Figure 1.** Classes Flat and Text are initially derived from File. Classes Exec and Tree are added dynamically.

COMMANDS, FLAT FILE SYSTEM			
Class	Command	Arguments	Effect
File	<b>list</b>		<i>identify this file</i>
	<b>parent</b>		<i>change current file to parent</i>
Text	<b>append</b>	<i>&lt;text-line&gt;</i>	<i>append &lt;text-line&gt; to this file</i>
	<b>print</b>		<i>print the contents of this file</i>
	<b>clear</b>		<i>empty this file</i>
Flat	<b>list</b>		<i>list files in this directory</i>
	<b>create</b>	<b>Text</b> <i>&lt;file-name&gt;</i>	<i>create a text file</i>
	<b>current</b>	<i>&lt;file-name&gt;</i>	<i>change current file to argument</i>

**Figure 2.** Each command is handled by a call to a member function of the same name.

```

Flat!root: create Text foo    a text file in this directory
Flat!root: current foo       make it the current file
Text!foo:  append This is foo append a line to it
Text!foo:  print              display its contents
This is foo
Text!foo:  parent              reset the current file
Flat!root:
  
```

### Linking New Code

The code for the flat file system is outlined in Figure 3. The dynamic linking facilities are accessed through an object of class Program; this class is declared in the library `dynamic.h`. Function `main` creates a program object, initialized with the name of the executing program (passed in `argv[0]`). It then creates the root of class Flat, initializes it, creates a reader object, and starts up the reader.

A command `load` for dynamic linking is recognized by class File; this command is in addition to those in Figure 2. Function `File::load` in Figure 3 is implemented by calling a member of the library class Program.

Function `Program::load` links new code into a running program. Like other incremental linkers, it works by allocating enough storage for the new code, invoking the UNIX system `ld` command (specifying incremental linking and the address of the storage) and reading the out-

---

<code>#include &lt;dynamic.h&gt;</code>	<i>declares class Program, Promotable</i>
<code>class File : public Promotable { ... };</code>	
<code>class Text : public File { ... };</code>	
<code>class Flat : public File { ... };</code>	
<code>class Reader { ... };</code>	
<code>Program *program;</code>	
<code>Reader *reader;</code>	
<code>int main(int argc, char *argv[]) {</code>	
<code>program = new Program(argv[0]);</code>	<i>object initialized with program name</i>
<code>Flat * root = new Flat();</code>	<i>create root file object</i>
<code>root-&gt;init("root");</code>	<i>initialize root</i>
<code>reader = new Reader(root);</code>	<i>reader, initialized with root</i>
<code>reader-&gt;startup();</code>	<i>prompt and read commands</i>
<code>return 0;</code>	
<code>}</code>	
<code>void File::load(String&amp; line) {</code>	
<code>...</code>	<i>extract object file name and libraries</i>
<code>program-&gt;load((UNIX object file), (libraries));</code>	
<code>}</code>	
<code>void File::become(String&amp; line) {</code>	
<code>FileObject[line]-&gt;promote(this);</code>	
<code>...</code>	<i>check whether promoted successfully</i>
<code>}</code>	
<code>void Flat::create(String&amp; line) {</code>	
<code>...</code>	<i>extract class name and file name</i>
<code>Text * n = TextObject[(class-name)]-&gt;mk();</code>	<i>create a text file</i>
<code>n-&gt;init((file-name), this, files);</code>	<i>initialize it</i>
<code>files = n;</code>	<i>a bookkeeping operation</i>
<code>}</code>	

**Figure 3.** The structure of the code for the flat file system. Dynamic linking facilities are accessed through commands `load` and `become`, which correspond to member functions of class `File`.

---

put of the linker into memory. These actions would be sufficient to accomplish dynamic linking in C, but in C++ it is also necessary to initialize any static objects in the new code. Initialization is arranged by running a version of the *patch* facility to find the constructors and then executing them at the end of the linking process.

Suppose that the compiled code for class `Exec`, derived from `Text`, appears in a UNIX system file `Exec.o`. This code is linked into the flat file system by the command

```
Flat!root: load Exec.o -lC          load an object file and its library
Flat!root:
```

The object file `Exec.o` contains the class object for class `Exec`, as a static member. When the class object is initialized during loading, it enters itself into the maps in the base classes

Text and File. Thus, this command makes class Exec known to the file system. In particular, when `<class-name>` denotes Exec, the expression

```
TextObject[<class-name>]
```

denotes the class object for class Exec. This expression is taken from the body of `Flat::create` in Figure 3.

Now, executable files can be created along with ordinary text files. In our toy file system, a file of class Exec can be executable or not, controlled by the command `setX`. If a file of class Exec is executable, the command `exec` sends its contents to the UNIX shell; otherwise the command evokes an error message. Of course, `exec` is not understood by any of the other classes.

The following commands create a file `bar`, append a line containing date, and then execute it as a UNIX system command:

<code>Flat!root:</code>	<code>create Exec bar</code>	<i>a new file in this directory</i>
<code>Flat!root:</code>	<code>current bar</code>	<i>make it the current file</i>
<code>Exec!bar:</code>	<code>append date</code>	<i>put some text into it</i>
<code>Exec!bar:</code>	<code>exec</code>	<i>try to execute it</i>
	<code>not executable</code>	
<code>Exec!bar:</code>	<code>setX 1</code>	<i>make it executable</i>
<code>Exec!bar:</code>	<code>exec</code>	<i>execute it</i>
	<code>Wed Nov 29 18:07:40 EST 1989</code>	
<code>Exec!bar:</code>		

### Motivation for Object Promotion

An object of a base class is said to be *promoted* to a derived class, if its type is changed to that of the derived class. Promotion is type safe, since the derived class supports all the operations of the base class. In order for promotion to be feasible in C++, the derived class must not declare any new data members, and it must not redefine any non-virtual function members of its base class. Object promotion is implemented by adjusting the *vptr* of the object to point to the *vtbl* of the target class.<sup>5</sup>

Object promotion allows an existing flat file system to be extended into a hierarchical one. The extension begins when we load file `Tree.o` containing the compiled code for class `Tree`:

```
Exec!bar: load Tree.o -lC
Exec!bar: parent
Flat!root:
```

Unfortunately, the root is of class `Flat`, which does not allow subdirectories to be created. This restriction is enforced by accessing object-creation functions through the class object of class `Text`. The body of `Flat::create` creates `bar` by executing lines equivalent to

```
Text *n = TextObject["Exec"]->mk();
n->init("bar", this, contents);
```

The version of `create` in class `Tree` uses the class object of class `File`, so it can create objects of any of the derived classes of `File`, including `Flat` and `Tree`. It executes

<sup>5</sup> Promotion is a type-safe special case of the unchecked "become" system primitive in Smalltalk, which "[swaps] the instance pointers of the receiver and the argument (Goldberg and Robson [1983])."

```
File *n = FileObject["Exec"]->mk();
n->init("bar", this, contents);
```

The following command promotes the root from class Flat to class Tree:

```
Flat!root: become Tree
Tree!root:
```

The prompt confirms the new class of root. The promotion is implemented by executing a line equivalent to

```
FileObject["Tree"]->promote(this);
```

with this pointing to root.

The file system is now complete. The remaining lines demonstrate that subdirectories can indeed be created.

```
Tree!root: create Tree sub           a subdirectory of the root
Tree!root: current sub              make it the current file
Tree!sub: create Text subfoo        create a text file in it
Tree!sub: list                      list its contents
    sub : Tree
    subfoo : Text
subfoo
Tree!sub: parent                    back to the root
Tree!root: list                     list its contents
    root : Tree
    sub : Tree
    bar : Exec
    foo : Text
Tree!root:
```

## 5. Class Objects and Their Classes

For every class that inherits from either class Class or class Promotable, our preprocessor automatically generates class objects and their classes; the preprocessor runs after cpp and before cfront. The class object inheritance hierarchy mirrors the class hierarchy. Multiple inheritance is supported.

### Class Object Members

The two main operations supported by class objects are table lookup and object creation. Both are used in the following statement, adapted from the file-system code. It creates an executable file:

```
File *n = FileObject["Exec"]->mk();
```

The operator [] is overloaded for table lookup, and mk is an object-creation function.<sup>6</sup>

The class object code generated for a class depends only on its constructors. The class object members in Figure 4 came from the code generated by our preprocessor for the following class:

<sup>6</sup> Presently, the preprocessor generates parameterless mk functions only. Another function must be executed to initialize the new object. In the future, we plan to generate a mk function for each constructor.



---

```

class FileClass : public ClassClass {
    __Class_Map *__map;
protected:
virtual File*      mkFile();
virtual Class*     mkClass();
public:
                    FileClass(const char *name);
    FileClass* operator[];
    File*      mk();
};
static FileClass FileObject("File");

```

Figure 4. Class object members generated for class File.

---

```

class File : public Class {
    ...
    File() {}
    ...
};

```

### Maps from Class Names to Class objects

Each class object is supplied the name of the class it represents. Thus, FileObject is supplied the string File. The initialization code for class objects uses the string to create and initialize a map from class names to class objects. The following program fragment approximates the initialization code:

```

__map = new __Class_Map;                create a map
__map->install(name, this);              name maps to this

```

The actual initialization code checks to see if a map already exists. The reason is that the FileClass constructor is called not only for FileObject, but also for the class objects of the derived classes. The use of an existing map ensures that later, when Text and Flat are derived from File, their names and class objects will be inserted into the existing map.

The existence of a map for a class is checked using an additional map, \_\_Class\_map, from class names to their maps. Thus, the following constructor code first looks up the map in the "meta-map," and then, in the looked-up map, installs the class name and this class object:

```

FileClass(const char *name): ... {      a class object constructor
    if(!(__map = __class_map.lookup("File"))) { if no map exists
        __map = new __Class_Map;          create it
        __class_map.install("File", __map); record its existence
    }
    __map->install(name, this);           map name to this
}

```

The table-lookup operator [] returns a pointer to the desired class object; otherwise, the lookup is unsuccessful, so it issues an error message and returns this class object itself:

```

FileClass* operator[](const char* __mc) {
    FileClass* __Class = (FileClass*)__map->lookup(__mc);
    return __Class ? __Class :
        ((*class_handler)("illegal mapping", __mc), this);
}

```

Whenever a class object function finds an error, it calls `class_handler` with an appropriate error message and a class name; a default handler is supplied.

## Object-Creation Functions

For technical reasons, the `mk` functions in a class object are not virtual functions. We discuss the main reason before examining the implementation of the `mk` functions.

In the file system example, a file of class `<name>` is created by executing an expression of the form

```
FileObject [<name>] -> mk()
```

Here, `FileObject` is the class object of the base class `File`. For the moment, constructor parameters will be ignored.

In more detail, we have

EXPRESSION	POINTS TO
<code>FileObject["File"]</code>	<code>FileObject</code>
<code>FileObject["Text"]</code>	<code>TextObject</code>
<code>FileObject["Exec"]</code>	<code>ExecObject</code>

Hence, the `mk` function for the class object of class `<name>` returns a pointer to a new object of class `<name>`. Since the corresponding classes `FileClass`, `TextClass`, and `ExecClass` are part of an inheritance hierarchy, we might expect their `mk` functions to be virtual, with the following definitions:

```

File *FileClass::mk() { return new File; }
Text *TextClass::mk() { return new Text; }
Exec *ExecClass::mk() { return new Exec; }

```

Unfortunately, C++ does not allow derived classes to redefine the return types of virtual functions, as needed for these definitions.<sup>7</sup>

The class declarations in Figure 5 avoid such type problems. The `mk` functions are not virtual; however, they call appropriate auxiliary virtual functions. With these declarations, the expression

```
FileObject [<name>] -> mk()
```

creates an object of class `<name>` as follows. Since `<name>` is not known at compile time, the type of

```
FileObject [<name>]
```

<sup>7</sup> Another reason for using non-virtual `mk` functions is to keep the `mk` functions for one class independent of the `mk` functions for other classes in its inheritance hierarchy. In this way, we can create a `mk` function corresponding to each constructor, without requiring derived classes to provide `mk` functions of the same type.

---

```

class FileClass : public ClassClass {
    File *mk() { return mkFile(); }
    virtual File *mkFile() { return (new File); }
};

class TextClass : public FileClass {
    Text *mk() { return mkText(); }
    virtual Text *mkText() { return (new Text); }
    virtual File *mkFile() { return (File *) (new Text); }
};

class ExecClass : public TextClass {
    Exec *mk() { return mkExec(); }
    virtual Exec *mkExec() { return (new Exec); }
    virtual Text *mkText() { return (Text *) (new Exec); }
    virtual File *mkFile() { return (File *) (Text *) (new Exec); }
};

```

Figure 5. Rather than creating objects directly, the mk function in each class calls a virtual function to do the job.

---

is pointer to FileClass. Hence, the call mk() is to FileClass::mk, which calls mkFile. By design, mkFile is virtual, so an object of the desired class is created.

For example, suppose that *<name>* is Exec. The expression

```
FileObject["Exec"]
```

points to ExecObject, despite the cast that converts its type to pointer to FileClass. The virtual function mkFile therefore denotes the version in class ExecClass, which creates an Exec object.

## 6. Discussion

Class objects are a type safe and flexible mechanism for creating instances of dynamically linked derived classes. They are generated by a preprocessor, which conforms to the C++ rules for class definition and linkage. This approach is therefore portable. After dynamic linking, construction of objects of the new classes requires a lookup through a simple map, and then the new objects are used in the normal way. Therefore, this approach is efficient.

The rest of this section touches on promotion and linking.

### Object Promotion

Every class that inherits from Promotable supports an operation promote, in addition to all of the operations discussed in Section 5.

Although promotion can be justified by saying that it converts an object from a supertype to a subtype — a safe direction — the operation is implementation dependent. It relies on the usual C++ layout of objects with a pointer to a table of virtual functions. By adjusting this one pointer, we change the binding of virtual functions. Such pointer-manipulation can be used to

implement promotion only when the data layout of the derived class is the same as that of the base class.<sup>8</sup>

Promotion can be useful however; the flat file system in Section 4 remains flat without it, even after class `Tree` is linked in.

### Linking: What We Did

The actual linking of new classes is system-dependent, since each system has a slightly different object format. For example, System V uses COFF (Common Object File Format), but BSD systems do not, and symbol relocation varies from machine to machine. Therefore, only the problems faced in linking will be mentioned here.

We used `ld`, the standard UNIX system linker, which commonly includes an option to link new code to an `a.out` file. The linking operation was put in a library to hide system dependencies. The operation is available as `Program::load`; see Section 4 for an example of its use.

Since C++ allows more complex global variable initialization than C, special constructor functions must be called when the program begins execution. When new code is linked, these functions have to be found, the constructors called, and the corresponding destructors stored for invocation at program termination. The C++ translator `cfront` comes in two versions, known as `patch` and `munch`, that differ only in the way they identify and call constructors and destructors. Correspondingly, we have two versions of our linker, although the `munch` version is not fully operational. Both versions could be tuned for speed.

When `load` is invoked on an instance of `Program`, the code is first linked by using `ld`. To properly link the code, the linker needs to know the space the code is going to reside in, so first space is allocated using `new` and then the linker is called. The constructors and destructors are then found, and the code relinked if necessary. Finally, the code is loaded into memory and the constructors called.

### Acknowledgments

The opening scenario and the file-system example are due to Sandy Fraser. We thank Jim Coplien, John Puttress, and Bjarne Stroustrup for helpful discussions. This paper grew out of a talk hosted by Rob Murray at Liberty Corner.

### References

- Call, L. A., Cohrs, D. L., and Miller, B. P. [1987]. CLAM — an open system for graphical user interfaces. *ACM SIGPLAN Notices* 22:12 (December), 277-286. OOPSLA '87 Proceedings.
- Chesson, G. L. [1979]. DATAKIT software architecture. *ICC '79, IEEE Intl. Conference on Communications*, 2, 20.2.1-20.2.5.
- Ellis, M. A., and Stroustrup, B. [1990]. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass. to appear.
- Fraser, A. G. [1979]. DATAKIT—a modular network for synchronous and asynchronous traffic. *ICC '79, IEEE Intl. Conference on Communications*, 2, 20.1.1-20.1.3.

<sup>8</sup> Future extensions can be anticipated by including a dummy pointer of type `void *` in the base class. The derived class can use the pointer to reach an extension object containing additional data members.

- Goldberg, A. [1984]. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Mass.
- Goldberg, A., and Robson, D. [1983]. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Mass.
- Koenig, A. R. [1988]. Associative arrays for C++. *USENIX Summer Conference* (June), 173-186.
- Puttress, J. J., and Goguen, H. H. [1986]. Incremental loading of subroutines at runtime. manuscript (April) AT&T Bell Laboratories, Murray Hill, N.J.
- Quong, R. W. [1989]. The design and implementation of an incremental linker. CSL-TR-88-381 (July) Computer Systems Lab., Stanford Univ.
- Sethi, R. [1989]. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, Mass.
- Snyder, A. [1986]. Encapsulation and inheritance in object-oriented languages. *ACM SIG-PLAN Notices* 21:11 (November), 38-45.
- Stroustrup, B. [1987]. Possible directions for C++. *USENIX C++ Workshop*, 399-416.
- Teitelman, W., and Masinter, L. [1981]. The Interlisp programming environment. *IEEE Computer* 14:4 (April), 25-33.



# FIELD Support for C++

Steven P. Reiss and Scott Meyers

Department of Computer Science  
Brown University, Box 1910  
Providence, RI 02912.

spr@cs.brown.edu and sdm@cs.brown.edu

## Abstract

FIELD is an open, language-independent programming environment offering a wide array of program development tools, including compilers, debuggers, build utilities, cross-referencers, profilers, call-graph displayers, and data structure displayers, as well as a uniform interface to source code via an annotation editor. Its most important tool specifically tailored to C++ development is `cbrowse`, a class browser that we believe to be the most sophisticated available. This paper is devoted primarily to describing the capabilities of `cbrowse`.

## Introduction

FIELD is an open, integrated programming environment built around software development tools running under Unix. Within FIELD, standard tools such as compilers, debuggers, profilers, etc., run as usual, but they are able to communicate with one another by passing messages through the FIELD message server, which categorizes incoming messages and forwards them to only those applications that have registered an interest in receiving messages of that category, a process known as *selective broadcasting*. This mechanism allows a user to, for example, add a breakpoint annotation to a program in FIELD's text editor, which results in the editor sending a message to the debugger informing it where the breakpoint should be set. Similarly, when the debugger arrives at a breakpoint, it sends a message to the editor telling it where it has stopped, allowing the editor to adjust its display to reflect the current location in the program (source file and line). Other publications [Rei90a, Rei90b] describe FIELD and selective broadcasting in detail; interested readers are referred to those publications for further details.

Most of FIELD's tools are independent of the source language, so they are just as useful with C++ as they are with C and Pascal, the other languages currently supported. In addition to the annotation editor, compilers, loaders, and debuggers, such tools include graphical build interfaces (front ends to utilities like `make` and `shape`[KLM<sup>+</sup>90].), call-graph displayers, data structure displayers, profilers, and cross-reference databases. Language-specific support for C++ consists of a powerful graphical and textual browser, `cbrowse`, as well as enhancements to the debugger interface. This paper describes `cbrowse` and the enhancements to the source-level debugger.

## `cbrowse`

For most people, the first thing that comes to mind when they think of browsers for object-oriented languages is a graphical display of the class hierarchy. For single inheritance, producing a pleasing graphical display is fairly straightforward, as tree layout is well understood. With multiple inheritance, the problem becomes that of laying out directed acyclic

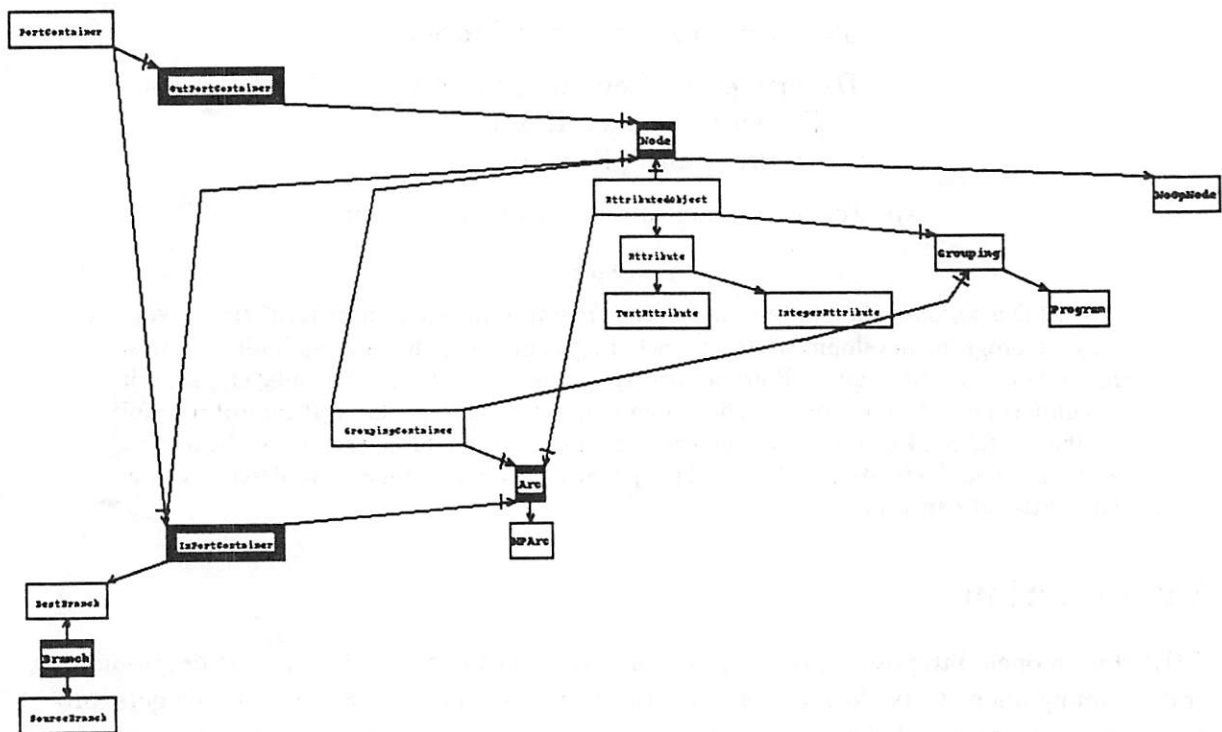


Figure 2: Inheritance Graph Using Grid Layout

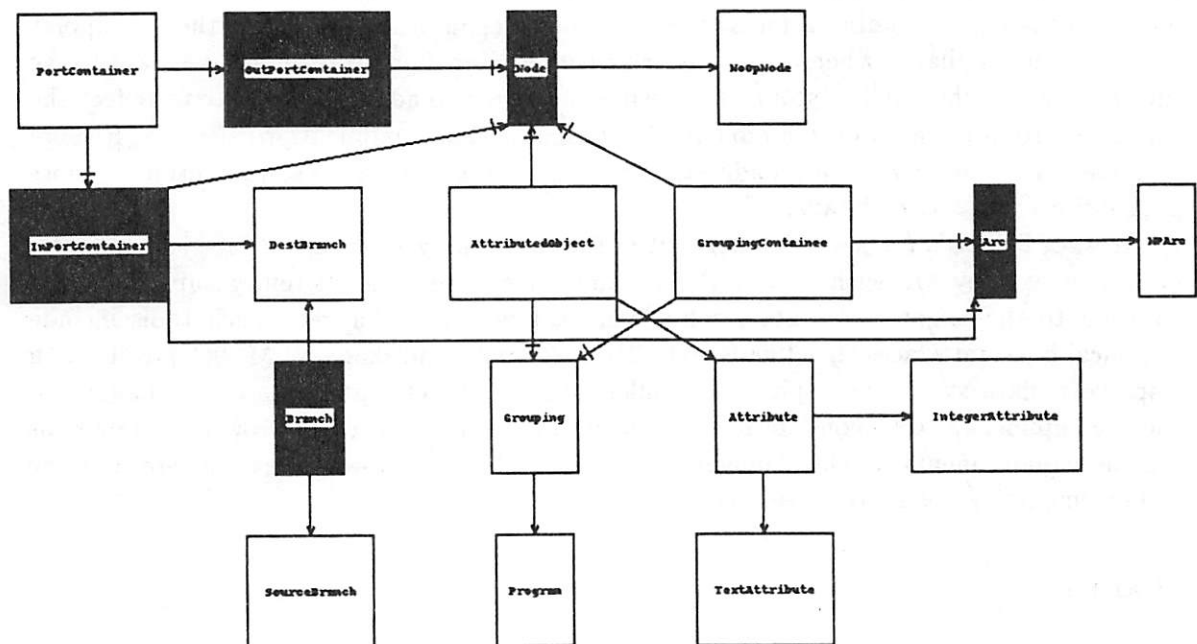


Figure 3: Inheritance Graph Using Breadth-First Layout

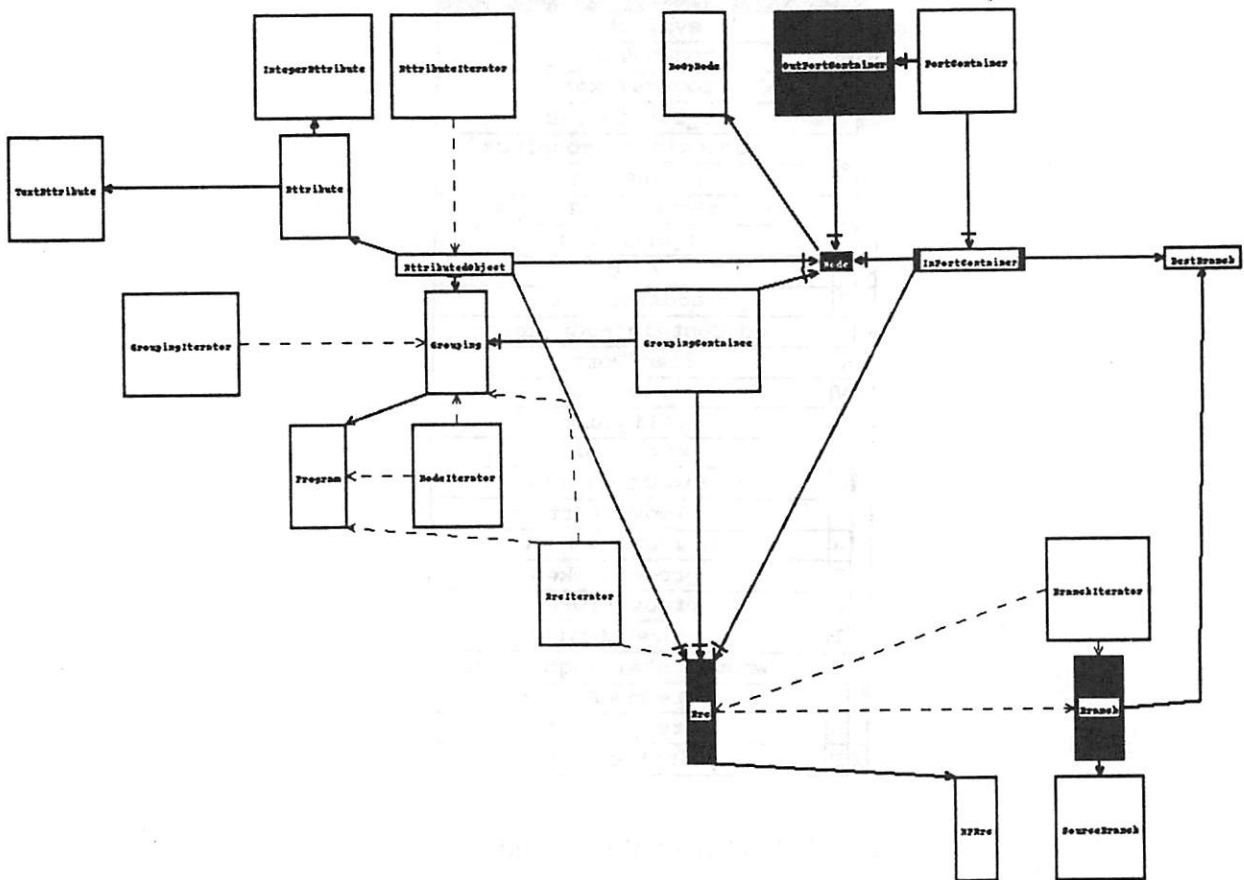


Figure 4: Inheritance and Friend Relationships

shows all the information for class *Node*. It is divided into three columns. The leftmost column indicates the accessibility level (private, protected, or public) of each member and differentiates data members from function members. The middle column provides status information for each member (e.g., whether it's constant, whether it's virtual, etc.), and the rightmost column gives member names and indicates whether they are inherited or defined locally. Details:

- **Leftmost column:** Each entry in this column is a triangle, facing either right or left. Right-facing triangles indicate functions, left-facing triangles indicate data members. The fill pattern of the triangle reflects the accessibility of the corresponding members: black is for private members, gray for protected members, and white for public members. Examples: `evaluate_` is a private pure virtual member function; `id` is a protected constant data member.
- **Middle column:** Meanings of status codes are given in table 1.
- **Rightmost column:** Members defined locally are listed as defined. Names of inherited members are preceded by a double-colon. If desired, users can have `cbrowse` also show the class from which a member is inherited.

Most of the time, users don't want to see all of this information. For example, potential class users will have little interest in private or protected fields, and potential class

Node	
P	evaluate_
vi	processToken_
vi	processToken_
	::_attributes
	::_containingGroupings
	::_inPorts
	::_inPortsWithTokens
	::_outPorts
C	::id
I	::addAttribute
I	::addContainingGrouping
V	::addPort
V	::addPort
ci	::eligible
V	evaluate
C	::findAttribute
	::lookupPort
ci	::numAttributes
I	::processToken
I	::processToken
I	::removeAttribute
I	::removeContainingGrouping
vi	::removePort
vi	::removePort
X F	printNodeInfo

Figure 5: Display of Member Information

developers have no reason to care about private fields. Some users won't care about friend relationships, and others will have no desire to see member properties (such as whether they are constant, static, inline, etc.). cbrowse allows users to toggle the visibility of all this information. In particular, users may choose to see or not see data members, function members, private members, protected members, public members, friend links, member access levels and member status data. They may also toggle the visibility of inherited members, thereby displaying either a complete class interface or only those aspects of the interface defined in the class.

More powerful elision commands are also available. Users may specify classes to include and/or exclude from the display using grep-style regular expressions, and may also employ regular expressions to specify which data and/or function members should be displayed. Regular expressions for determining member visibility may be applied on a global basis or

Code	Meaning
V	Virtual function
P	Pure virtual function
I	Inline function
F	Friend function
C	const member
S	Static member

Table 1: Codes for Member Status Information

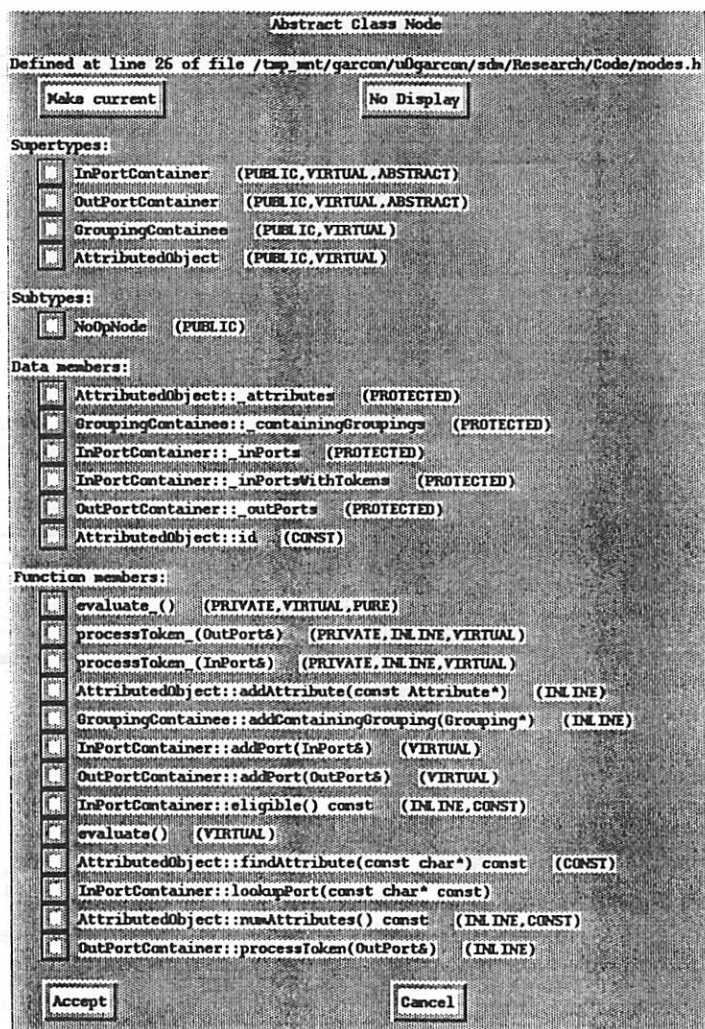


Figure 6: Textual Class Summary

to only those classes matching a given regular expression.

Clicking on a member field can bring up a window giving textual information about the member. This information includes the source file and line of the member's declaration, its access level, its properties, and, for functions, its signature. Similar information can be obtained about a class by clicking on the class, as shown in figure 6. It is also possible to open a class information window that will always display a textual summary of the currently selected class. If FIELD's annotation editor is being run at the same time as cbrowse, it will automatically bring up the appropriate source file at the line of declaration of the selected class or member.

When inherited members are being shown, selecting an inherited member not only highlights the member, it also highlights the original point of definition, the path by which the member was inherited through the class graph, and the locations "below" the member where the definition continues to be inherited. This facility is shown in figure 7. In the figure, all classes except *AttributedObject*, *Node*, and *NoOpNode* have been removed from the display. The currently selected class is *Node*, hence its depiction in reverse video, and the currently selected member is *FindAttribute*, which is shown highlighted in gray. *FindAttribute* is defined in class *AttributedObject*, so it is highlighted there, too, and the



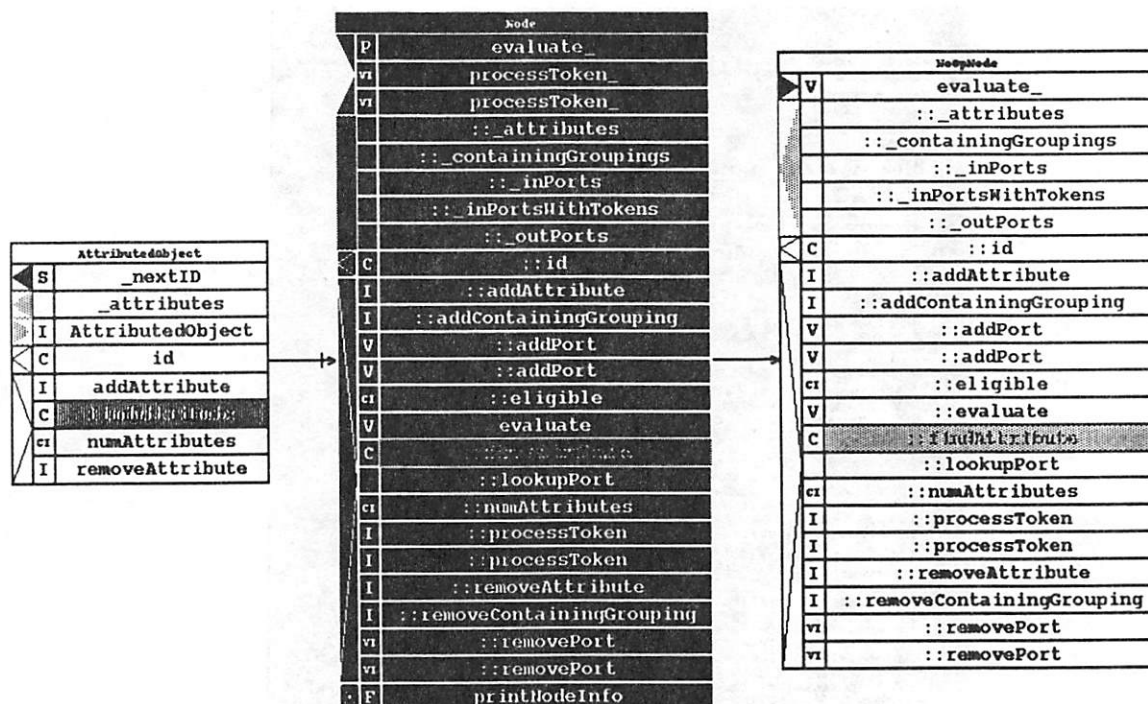


Figure 7: Display Showing Extent of Inherited Member

definition continues on “through” *Node* to class *NoOpNode*, so it is also highlighted there.<sup>2</sup> Thus, with a single glance at the graph, a programmer can determine where a particular definition comes from and the extent of its influence. This can be useful in a variety of ways. For example, it can be used to ensure that a virtual function is redefined in all subclasses (if indeed that is supposed to be the case). Alternatively, it can quickly reveal whether a member designed to be inherited has been redefined in subclasses.

During program execution within FIELD, cbrowse dynamically highlights the member functions being executed. This allows for a simple form of program visualization.

## Other FIELD support for C++

FIELD provides uniform interfaces, both graphical and textual, to the debuggers dbx and gdb, and performs automatic name mangling on input and name demangling on output for C++ programs. To the usual debugger command set it adds a new command that reveals the dynamic type of an object pointed to by a pointer or referenced by a reference. This allows variables to be completely characterized inside the debugger, something that is not possible with standard dbx or gdb. The FIELD debugger interfaces also allow the specification of breakpoints via regular expressions, e.g., it is possible to break on entry to any function matching a given regular expression. This facility is convenient in its own right, but it is particularly useful for setting breakpoints in virtual functions, which are bound to call sites

<sup>2</sup>On a color display, the currently selected class and member would be displayed in colors specified for this purpose. There are actually five such colors: one for the currently selected class; one for the currently selected member; one for the original point of definition of the current member; one for members “between” the original definition and the current member; and one for members “beyond” the current selection that share the same definition. Such colors are configurable at initialization by the user.

only at runtime.

All the standard FIELD tools may also be used with C++ , including build utilities, cross-referencers, textual and graphical program profilers, call-graph displayers (with dynamic highlighting), data structure displayers (with dynamic highlighting and modification), and the annotation editor.

## Status and Availability

FIELD is a rapidly evolving environment, with new tools being added on a regular basis, and functionality being added to existing tools as users request it. For example, all the C++ support tools have been developed within the past nine months, including `cbrowse`, the new debugging commands, and the necessary parsing and name mangling/demangling software on which they depend. The graphical build utility interface is the most recent addition to FIELD, with the first prototype only recently having become functional.

FIELD itself is based on Unix, the X Window System, and the tools of the Brown Workstation Environment[RS89]. It is currently available on Sun workstations running SunOS 3.x and 4.0.x, and on DEC DECstations and Microvaxes running Ultrix. It may be obtained by sending electronic mail to one of the authors or by requesting a software distribution form from the Brown University Computer Science Department.

Comments and suggestions from users of FIELD and/or BWE are actively solicited. Such feedback frequently results in the addition of valuable new features to the software, often quite quickly after the feedback is received.

## Acknowledgments

Support for this research was provided by the NSF under grant DCR 8605567; by DARPA under contract N00014-83-K-0146, ARPA order 6320; and by the Digital Equipment Corporation under agreement 393.

## References

- [KLM<sup>+</sup>90] Wilfried Koch, Andreas Lampen, Axel Mahler, Wolfgang Obst, and Uli Pralle. A Tutorial Introduction to the shape-Toolkit. Draft from the Technische Universität Berlin, January 1990.
- [Rei90a] Steven P. Reiss. Connecting Tools using Message Passing in the FIELD Program Development Environment. *IEEE Software*, July 1990. Also available as Brown University Computer Science Department Technical Report CS-88-18, "Integration Mechanisms in the FIELD Environment," 1988.
- [Rei90b] Steven P. Reiss. Interacting with the FIELD Environment. *Software: Practice and Experience*, 1990. To appear.
- [RMD89] Steven P. Reiss, Scott Meyers, and Carolyn Duby. Using GELO to Visualize Software Systems. In *Proceedings of UIST '89*, October 1989.
- [RS89] Steven P. Reiss and John T. Stasko. The Brown Workstation Environment: A User Interface Design Toolkit. In *Preprints of the IFIP WG2.7 Working Conference on Engineering for Human-Computer Interaction*, August 1989.



# Customization in C++

*Douglas Lea*

SUNY at Oswego / NY CASE Center / GNU Project  
(dl@oswego.edu)

## Abstract

Extensions to C++ are proposed that would enable the directed synthesis of code as an alternative to dispatch-based code generation. The strategies are similar to those used in the SELF compiler, which uses customization techniques to impressive effect. The language extensions required to support customization also enhance support for other uses of inheritance in C++, and are useful in conjunction with proposed additions of parametric types.

## 1 Introduction

One of the smallest, but most important features introduced in C++ 2.0 is improved support for abstract base classes (ABC's) via the use of "pure virtual" member functions. For example,<sup>1</sup>

```
class Matrix
{
public:
    virtual      ~Matrix() {}
    virtual int   rows() const = 0;
    virtual int   cols() const = 0;
    virtual float elem(int i, int j) const = 0;
    int           size() { return rows() * cols(); }
};
```

This declares a Matrix class in terms of its protocol or interface (i.e., the signatures of member functions), but not in terms of member function implementations or the data representations upon which they operate. Subclasses of Matrix are required to define the implementations of these functions themselves. As is often useful, this ABC also declares a non-pure member function (`size`) that operates exclusively via the pure virtual functions. Of course, it is also possible to declare ordinary "client" functions that operate on any Matrix via these member functions, regardless of implementation. For example.

<sup>1</sup>I will use Matrix-based examples throughout the paper, since they serve well to illustrate various points. The examples are mostly realistic, but substantially stripped-down from those suitable for serious application. Just to stave off controversy, I will note a few of the most egregious simplifications. But in all other respects, this paper has little to do with Matrix classes per se.

```
float sum(const Matrix& m)
{
    float s = 0;
    for (int i = 0; i < m.rows(); ++i)
        for (int j = 0; j < m.cols(); ++j)
            s += m.elem(i, j);
    return s;
}
```

ABCs are closely related to the concept of "Abstract Data Types", or ADTs (see [7]), in that they completely hide from clients of the class all representation and implementation details (as defined in any given implementation subclass). They are not completely isomorphic to ADTs, since there is no C++ support for semantic specification of such classes (but see [5]), and since ABCs are declared and used in an object-oriented, rather than algebraic fashion.

Support for ABCs completes the possibilities implicit in the notion that protocol inheritance (also known as "specification inheritance", "interface inheritance", or just "subtyping") is a logically distinct concept from implementation/representation inheritance. Discussions of the importance of separating these two roles of inheritance may be found in [1, 10]. In C++, one can now obtain protocol-only inheritance via ABCs, implementation-only inheritance via "private" subclassing or simple composition, or both, via non-abstract "public" subclassing.

The availability of ABCs encourages a style of programming in which every different set of protocols gets its own ABC, and every implementation strategy gets its own subclass. Classes that define both protocol and implementation are needed only when specifications are narrow enough to dictate particular representations. Often, specifications only partially restrict implementations, leading to partially abstract classes. (For example, it might be defensible to force all matrices to possess member variables `int r, c;` to hold the numbers of rows and columns, and to rewrite `rows()` and `cols()` accordingly.)

C++ support for ABCs considerably enhances creation of both domain-independent and domain-specific reusable packages and libraries:

- ABCs support the design of class hierarchies without forcing premature commitments to implementation details.
- ABCs improve support for guaranteed "plug-compatible" implementations. Clients may replace one implementation subclass with another, without any other program changes.
- ABCs allow simpler integration of classes and libraries from different sources, by allowing integrators to subsume two different implementations under a common ABC.
- ABCs improve prospects for standardization efforts, since they address only specification, not implementation issues.
- ABCs are vital for development of packages in which subclasses with common protocols require vastly different implementations.



This last point is the most important one for a large number of applications. Consider the need for an integrated linear algebra package based in part on class `Matrix`. In order to be maximally useful, one would need a large number of `Matrix` subclasses, including `RowMajorMatrix`, `ColumnMajorMatrix`, `TridiagonalMatrix`, `RowSparseMatrix`, and dozens of others, as well as additional intermediate abstract subclasses, to support the range of algorithms that have been developed for dealing with such special (in terms of either representation details or element structure) `Matrix` types. For example,

```
class DenseMatrix : public Matrix
{
public:
    float& operator () (int i, int j) = 0;
    virtual DenseMatrix& operator += (const Matrix& b)
        { /* ... */ return *this; }
};

class RowMajorMatrix: public DenseMatrix
{
private:
    int r; int c; float* d;
public:
    RowMajorMatrix(int m, int n);
    RowMajorMatrix(const RowMajorMatrix& b);
    ~RowMajorMatrix();
    int rows() const { return r; }
    int cols() const { return c; }
    float& operator () (int i, int j)
    {
        if (i < 0 || i >= rows() || j < 0 || j >= cols())
            abort(); // or, someday, raise an exception
        return d[i * cols() + j];
    }
    float elem(int i, int j) const { return (*this)(i, j); }
};
```

There is a great deal of design elegance to be had in this approach. Unfortunately, there is also a great deal of inefficiency in the corresponding code generated by current C++ compilers.

Consider a call to `sum(a)` for a `RowMajorMatrix` `a`. Because all of the invoked operations are virtual, actual computation is swamped by virtual function calls. Worse, because the implementation code lying inside the virtual functions is opaque to the C++ compiler, none of the standard optimization strategies (like strength reduction, common subexpression elimination, and even automatic vectorization) that can be applied to array processing code inside good optimizing compilers are applicable. Perhaps the easiest (but *not* the most important) example of the kinds of problems encountered is illustrated in the index checking occurring inside `RowMajorMatrix::operator()`, where the checks must be performed on each `elem` call since the compiler cannot “see” the bounds checking code and statically

determine that all accesses within `sum` are legal, and thus optimize the checks away completely. For such reasons, this code is likely to execute around ten times more slowly on most machines than it would without the virtual calls. (Simple informal tests confirm the order of magnitude of this estimate.)

Array processing code represents an extreme, but very important case in which allowing a compiler to perform inline substitution of a few critical "lightweight" member functions, and then to procedurally integrate and optimize the resulting code represents the difference between merely attractive class designs and usable packages. The problem is not that these calls are virtual, or even that there is procedure call overhead at all. It is the failure to procedurally integrate code that most hampers performance: In the ideal case, the executable code produced for `sum` should be so well integrated that few machine operations could be said to "belong" to `sum` or the invoked member functions per se.<sup>2</sup> The failure of current C++ compilers to make powerful optimization strategies even conceivable is surely unacceptable to many potential users of classes like `Matrix` that are targeted for use in high-performance applications. Without accommodations, authors of such packages must sacrifice object-oriented design methodologies in favor of less reusable, reliable, and coherent ad hoc coding strategies.

The remainder of this paper proposes one solution to this mismatch between positive design benefits and negative implementation costs of ABC's, and inheritance in general.

## 2 Compiling polymorphic functions

Any class member function or top-level client function that *invokes* a virtual class function is polymorphic (or more precisely, "subtype polymorphic" [2]). Any such function, like `sum(const Matrix&)`, may be thought of as a stand-in for an unknown number of specific functions `sum(const RowMajorMatrix&)`, `sum(const TriDiagonalMatrix&)`, and so on.

There are two approaches for implementing functions that share the exact same "high-level" C++ source code.

In the standard "dispatch" approach normally employed by C++ compilers, all versions share the same actual machine code, but produce different run-time consequences because the indirect calls generated from the member function dispatch tables ("vtables") invoke different implementation code for different objects passed in as arguments (including `this` as an implicit argument for member functions).

A second approach is to synthesize, or *customize* as many different specific versions of a function as are actually needed in a given user program. Each customized version can then hard-wire code specifically geared for the given arguments by directly calling and/or procedurally integrating invoked member functions, rather than dispatching them. Thus, even though such polymorphic functions share high-level source code, they are implemented via different machine code. A compiler may implement this strategy via analogs of macro-expansion: The high-level source code may be recompiled as a new function in light of the

<sup>2</sup>The C underpinnings of C++ present several additional well-known obstacles (e.g., uncontrolled aliasing) for compilers attempting to perform powerful Fortran-like optimization strategies for numerical and array processing. These problems also require careful attention, but are not considered here.

actual arguments supplied to it. This is roughly similar to how call-by-name functions are compiled in some languages.

Both of these approaches have advantages and disadvantages that reflect classic compilation versus execution and time versus space tradeoffs. The dispatch approach is much less space-intensive, and easier to implement, since all versions of a polymorphic function share the same machine code. The customization approach may generate more efficient implementations at the price of potential code explosion.

There is an overriding reason that C++ compilers must support the dispatch model. Dispatching may be used when the exact run-time type of an object is not known internally to the compiler. Customization is impossible in such situations. This includes, especially, C++ programs in which heterogeneous collections of objects, all of the same base type but different exact types, are used as arguments to functions. Such programming practices are typical, but by no means universal in C++. In many applications, it would be considered a design or implementation flaw to have *any* run-time type uncertainty in a C++ program.

Clearly, when the exact types of argument objects are known, customization can remedy the kinds of performance problems seen with Matrix classes. Since particular versions of the function are generated, each can integrate the appropriate inlines, and optimize accordingly. This strategy has the potential of resulting in machine code as fast as obtainable in any language.

Customization, or code synthesis, is not really a new idea in C++. All operations on builtin types (`int`, `float`, etc.) are open-coded, or customized for the sake of efficiency, as are non-virtual inlines. Also, `X(X&)` constructors and assignment operators are automatically synthesized if not otherwise defined by programmers.

It is possible to extend customization to apply to arbitrary classes and functions. Ungar and Chambers [3, 4] discovered that dispatching and customization can happily coexist in compilers for object-oriented languages. Their SELF compiler automatically customizes some code, especially for "lightweight" objects like small integers, partially customizes other functions by hard-coding cases for only the most typical arguments, and uses dispatching elsewhere. As they mention, these compilation strategies are adaptable to other object-oriented languages, including C++.

Given the overall structure of C++, *automatic* customization does not seem to be an attractive option. The need for customization is primarily a pragmatic concern of the intended *applications* of classes and their supporting functions, not the classes themselves, and certainly not the compiler per se. Many users are perfectly content with fully dispatched functions, especially for prototyping, debugging, and other non-time-critical use. More typically, customization should be applied selectively. For example, speedups via customization might be of overriding importance in operations that multiply matrices, but not in routines that print out their contents. This is the same pragmatics problem underlying C++ support for inlining. Class users ought to have at least as much to say about whether functions are inlined as do class authors. C++ compilers need to make such decisions easier to implement than requiring users to edit source code. Therefore, if customization is supported in C++, compilers should provide means for end-users to control the degree that both customization and inlining are exploited in particular programs. The logistics of doing so are neither simple nor impossible.

### 3 Implementing Customization

The most straightforward way to implement customization is by first introducing a type qualifier for objects that is roughly analogous to the use of `inline` for functions. In fact, simply extending the applicability of `inline` to objects would suffice, except that there are cases where the two usages would conflict. However, as discussed further below, the already-reserved word `template` fills this role nicely, and has the right intuitive meaning.

As a type qualifier, `template` should be legal wherever `const` may be used. It may modify any variable, function parameter (including the implicit `this` parameter to member functions), or function return value designating a value (i.e., distinct object), reference, or pointer. The semantics of `template` are that

- The corresponding given type is not necessarily the most specific run-time type possessed by the declared object;
- The compiler should determine the actual type as the most specific type knowable at compile time from the object's initializer;
- If possible in light of this type information, the compiler should create a special version of the function in which this declaration is embedded, based upon knowledge of the actual type.

For example, redeclaring `sum(template const Matrix& a)` instructs the compiler to construct a special version of `sum` for each actual `Matrix` subtype for which it is called.

Even disregarding the customization aspects of the `template` modifier, there is ample justification for its introduction. It syntactically distinguishes otherwise different roles of programmer-provided type information in C++. Currently, any base-class *reference* may be used to refer to a single object of any subclass, and thus has its type implicitly fixed upon initialization. A base-class *value* may refer only to a particular class, never a subclass. And a base-class *pointer* may be used ambiguously to denote objects of any subclass, perhaps including objects of different subclasses over time, except when designating elements of an array, in which case a pointer is considered to have an exact type. Under this proposal, `template` may modify all three to clarify and extend the role of type information now associated with references.

To push this point a little further, an argument could be made that an additional type qualifier is required, say, `exact`, that restricts particular pointers and references to denote objects of one exact type only, as do non-template values. In this manner, one could merely say that reference declarations default to non-customized `template`, values to `exact`, and pointers to their current dangerously indeterminate role.<sup>3</sup> There are several other desirable results, especially in terms of enhanced type-safety, that would stem from support of `exact` references and pointers. This issue is touched on later, but is not central to the remainder of the proposal.

---

<sup>3</sup>Except that it would confusingly overextend their meanings, the keyword choice of `public` versus `private` instead of `template` versus `exact` would be appropriate.

In addition to an internal analog of a macro-expander, the major burden placed on a compiler supporting customization is the need for improved type-monitoring in order to ensure that the “best” versions of functions are synthesized and called. Two basic C++ constraints make this easier than it might be otherwise:

- Every object and expression has a most-specific type. For example, if `m` is declared as a `RowMajorMatrix`, then `m` has types `RowMajorMatrix`, `DenseMatrix`, and `Matrix`, with `RowMajorMatrix` being the most specific.
- All type-propagation is bottom-up. That is, the most specific types of the arguments to any operator or function fully determine the most specific type of the result.

Using these two constraints, customizing compilers can maintain the most specific types of all objects encountered in a program, across situations in which they are not currently required to do so.

There are many additional usage and implementation considerations. The most important ones are described here. A few details are omitted for the sake of brevity.

### 3.1 Inabilities to customize

As discussed above, there are cases where insufficient information is present to customize a call at compile time. Among the simplest examples is,

```
void f1()
{
    RowMajorMatrix a(10, 10);
    ColumnMajorMatrix b(10, 10);
    float s = sum( (rand() % 2) ? a : b );
}
```

The most specific type of the conditional is just `Matrix`, so a dispatch-based call must be made. Of course, more heroic intervention is certainly not disallowed: A compiler may, for example, safely convert this expression into `(rand() % 2) ? sum(a) : sum(b)`, thus allowing customization. However, these kinds of evasions are rarely possible in general.

### 3.2 Overriding automatic customization

It is not at all necessary for the compiler itself to synthesize specialized functions. As is already true in C++, users may override functions with more specific versions themselves. For example,



```
float sum(template const DiagonalMatrix& m)
{
    float s = 0;
    for (int i = 0; i < m.rows(); ++i) s += m.elem(i, i);
    return s;
}
```

The `template` here allows customization even if subclasses of `DiagonalMatrix` are introduced.

Existing argument-matching rules may be used in deciding when to call a synthesized version of `sum(DiagonalMatrix&)` rather than of `sum(Matrix&)`. As is true even now, when sufficiently precise type information is lacking, the most general versions of functions must be invoked/synthesized. This has important implications for the authors of specialized top-level or member functions: While such functions may be implemented arbitrarily differently than the base versions, they must preserve the same semantics, since authors cannot guarantee which version(s) will be called in a given program.

### 3.3 Custom return values

Type monitoring is more difficult for a compiler in the case of customized return values, since the exact return type is not known until *after* a customized function is synthesized. For the simplest example,

```
template Matrix& noop(template Matrix& a) { return a; }

void f2()
{
    RowMajorMatrix a(10,10);
    template Matrix& b = noop(a);
    float s = sum(b);
}
```

The most specific type of `b` is known to be `RowMajorMatrix` only after the specialized version, `noop(RowMajorMatrix&)` is synthesized.

### 3.4 Inheriting return value types

C++ currently possesses a serious, but removable limitation that often impedes type monitoring. For example, the declaration of `DenseMatrix& DenseMatrix::operator += (const Matrix& b)` above requires all subclass versions of `operator +=()` to return references to objects whose type is merely known to be some subclass of `DenseMatrix`. Thus any usage of this operator loses valuable type information for its result.

However, there is no convincing reason for this restriction. As discussed by Cook [6], it is entirely type-safe to allow subclass versions to return references to any subclass of `DenseMatrix`. With the `template` modifier, it becomes possible to express and support this.

```

class DenseMatrix
{
    //...
    virtual template DenseMatrix& operator += (const Matrix& b) template
        { /* ... */ return *this; }
};

```

The second use of `template` in the syntactic position (like `const`) that modifies the type of `this` is necessary here to properly propagate the type of `this` to the return value. Note also that a third use of `template`, to modify argument `b` might be applied here as well.

### 3.5 Custom values

While all examples so far have involved customization around references or pointers to objects, the entire proposal extends naturally to include customization around values. For example, for a `RowMajorMatrix a`; invoking a call-by-value version of `float sum(template Matrix m)`, via `float s = sum(a)`; matches argument `m` as a `RowMajorMatrix`, created via the `RowMajorMatrix X(X&)` constructor. The most specific applicable `X(X&)` constructor must be chosen for value copies of `template` arguments.

Support for “constructive” functions with customized return types is among the most attractive assets of the entire proposal. For example,

```

template DenseMatrix operator + (const template DenseMatrix& a,
                                const template Matrix& b)
{
    template DenseMatrix c(a); c += b; return c;
};

```

defines a version of `operator +()` applicable for any two appropriate arguments, without requiring case-by-case definition of all possible combinations of parameters, as would be required without customization. (Of course, in this example, it probably *would* be desirable to override this version for at least some argument pairs.)

There is, however, a significant problem encountered with compiling customized values that is not present for references or pointers. Consider,

```

void f3()
{
    RowMajorMatrix a(10, 10);
    ColumnMajorMatrix b(10, 10);
    template Matrix c = (rand() % 2)? a : b;
    //...
}

```

Here, the exact type of `c` is not known at compile time.

One could refuse to “unify” types in conditionals and related points of uncertainty when dealing with values, in which case this construct would be considered an ambiguity error.

A more aesthetically pleasing approach, that is in keeping with the idea that all differences between values and references should be based solely upon whether or not objects are copy-constructed, is to allow such compile-time uncertainties, and to resolve them at run-time. Doing so has some cost to the compiler. If the decision is delayed until run-time, then the size and initializer of `c` are not statically known, so its space must be allocated and deallocated from the freestore at run-time. However, there is no actual cost to the user. Without this support, if programmers need to create `c` in this fashion, then they must do so manually via pointers, as in

```
void f4()
{
    RowMajorMatrix a(10, 10);
    ColumnMajorMatrix b(10, 10);
    Matrix* c = (rand() % 2)? new RowMajorMatrix(a)
                           : new ColumnMajorMatrix(b);
    //...
    delete c;
}
```

So the net result is merely added convenience, and one less opportunity to (mis)use pointers, at the expense of compiler complexity. This appears to be a small price to pay for such a gain in utility and conceptual simplicity.

There is also a more pragmatic reason for desiring this strategy. If customization is disabled by a user who is not interested in potential speedups, but only the other semantic properties of the `template` modifier (say, in the case of `operator +()`, above), then a compiler would need to fall back on run-time freestore allocation in order to support this anyway. (It would also need to include function pointers to constructors in vtables, which is not currently typical.)

### 3.6 Custom constructors

If `template` may modify this for member functions, then customized constructors are immediately definable. For example,

```
class DenseMatrix
{
    virtual void copy(const Matrix&) = 0;
public:
    DenseMatrix(const Matrix& b) template { copy(b); }
    //...
};
```

This allows a base class to specify the general form of subclass `X(X&)` constructors, thus generalizing the current rules for automatic constructor synthesis, and clarifying the sense

in which constructors may be said to be inheritable (i.e., only via synthesis). Of course, these constructors may still be overridden in specific subclasses. Similar remarks hold for `X::operator = (X&)`.

### 3.7 Member variables

Consider.

```
class Z
{
private:
    template Matrix a;
    template Matrix& b;
public:
    Z(template Matrix& p, template Matrix& q) :a(p), b(q) {}
    //...
};
```

There are two problems encountered in supporting such custom member variables. The first is the value versus reference issue again: because the exact type of `a` differs across instances of `Z`, `sizeof(Z)` is not compile-time determinable across all `Z`s. This is remediable in the same manner as above, via internal fall-backs to freestore allocation and the internal use of a (fixed-size) pointer to the variably-sized part. Again, doing so results in no real cost to users who must otherwise simulate this effect manually.

The second problem is an inherent limitation of customization. Since customization revolves around classes, not individual objects, the present proposal does not fully enable compilers to hard-wire calls to either `Matrix` operations on either `a` or `b` inside `Z`'s member or user functions. Without heroic efforts, the customization benefits of `template` are lost: compilers are forced to disregard the specific object-by-object type information, and generate most-general-case dispatch-based code.

Neither of these issues is a compelling reason not to support the construct. However users must be made well-aware of its limitations.

### 3.8 Completely customized objects

Whenever a compiler can statically determine that *all* uses of an object have been customized (i.e., that no dispatch-based usages occur), then there is no need to allocate a dispatch table pointer ("vptr") for that object. This is only a significant issue for extremely lightweight objects, like instances of a `class Int` (not builtin type `int`). The advantages of programming with "simple" first-class objects without paying undue time or space penalties are highly desirable in many applications, making the corresponding sophisticated and challenging compilation analyses that would be required well worthwhile.

## 4 Customization and parametric types

Many readers will have noted that customization addresses similar issues, settled via similar implementation mechanisms (based on an internal macro-expander integrated into compilers), as were discussed in Stroustrup's [11] "Parameterized Types" (PT) proposal. While this is true, the two proposals are focused on two very different issues. Customization improves C++ support for emerging uses of inheritance, while PT extends C++ to overcome limitations that are logically distinct from subclassing.

In order to see why both PT and customization are needed, consider the central concern of the PT proposal, improved support for the definition and use of homogeneous container classes. Homogeneous container classes are structures (e.g., stacks, sets, matrices) that hold collections of other objects (e.g., ints, Complexes, Windows) but all of the exact same type per container class. (In contrast, heterogeneous container classes are those which hold collections of objects with the same base type, but possibly different exact types. Heterogeneous containers are readily implementable in C++, usually via pointer collections, without either PT or customization.) Homogeneous container classes are valuable because they enhance type-safety, because they generally support more efficient implementation than do heterogeneous containers, and because they provide mechanisms for ensuring that groups of objects possess coexistent lifetimes [8].

Customization in itself does not improve support for homogeneous containers because definitions of homogeneous container classes fall outside of the standard subtype/subclass hierarchy. For example, even though an `intStack` and a `WindowStack` share high-level specifications and code at some abstract level, they are not related via a common superclass.<sup>4</sup> This stems from the fact that while `intStacks` have member functions like `void push(const int&)`, `WindowStacks` have `void push(const Window&)`. Their specific argument types do not conform to each other, or to those of any other possible superclass. As discussed by Cook [6], in order to ensure complete type-safety, subclass virtual function arguments may not be restricted solely to argument types that are more specific than those of their parent classes. (Note that this problem is carefully avoided in section 3 above).

Stroustrup's solution to this problem was to introduce support for class templates, which serve as generators of full-fledged classes. Customization is fully compatible with the class templates in the common case where both are desirable. For example,

```
template <class T> class Matrix
{
    virtual      ~Matrix() {}
    virtual int   rows() const = 0;
    virtual int   cols() const = 0;
    virtual T     elem(int i, int j) const = 0;
    int           size() { return rows() * cols(); }
};
```

---

<sup>4</sup>Interestingly, in the singly-rooted class hierarchy of Smalltalk and related languages, a heterogeneous `ObjectStack` defined via root class `Object` is a *subclass* of all homogeneous `Stacks`. I know of no object-oriented language that supports easy expression of this fact.



```

template <class T> T sum(template const Matrix<T>& m)
{
    T s = 0;
    for (int i = 0; i < m.rows(); ++i)
        for (int j = 0; j < m.cols(); ++j)
            s += m.elem(i, j);
    return s;
}

```

This allows the code written for `sum` to be reused for matrices of any subclass of any `Matrix` class holding any numerical element type.

There may be ways to introduce both customization and parameterization into C++ that more seamlessly integrate the two concepts (as might possibly be done via an analog of BETA's "virtual classes" [9]). However, it is preferable to keep these mechanisms distinct: When you need one, having only the other doesn't help much. Since class templates generate several otherwise subtype-unrelated classes and/or subclasses simultaneously, they cannot be used to implement customization, or vice-versa, even both might rest upon the same internal compiler features.

There is another mesh point between PT and the present proposal. The class template mechanism does not improve one security aspect of homogeneous container classes. Consider the following example:

```

class B          { int x; /* ... */ };
class D : public B { int y; /* ... */ }

template <class T> class Stack100
{
private:
    int sp;    T data[100];
public:
    void push(const T& a) { data[sp++] = a; }
    T& top()             { return data[sp-1]; }
    //...
};

void f5()
{
    Stack100<B> s; D d; s.push(d); // ...
}

```

This implementation fails to meet one of the most important specifications of a stack: Given any sensible definition of "equals", and ignoring the lack of error handling in the simplified implementation, it is desirable, for any legal argument `a`, that after `s.push(a)`, `s.top()` should equal `a`. This property cannot hold when `d` is pushed in `f5()`. The problem is solvable via the `exact` qualifier described above. If the member functions could be declared as `void push(const exact T& t)` and `exact T& top()`, then legal arguments

could be restricted to those for which this promise can be kept. While the mechanisms described in the PT proposal are only tangentially relevant to this issue – the same problem would have arisen in defining a BStack100 directly – they do widen the impact of this limitation.

A final similarity between customization and PT lies in the extent to which both stretch the C-based separate compilation model up to, and probably past, its limits. While it may be possible to maintain the C model, both proposals would be best implemented under more sophisticated compilation management environments.

## 5 Acknowledgements

Thanks to Michael Tiemann, Doug Schmidt, Rajendra Raj, and Marshall Cline for commenting on drafts of this paper. This work was facilitated by an equipment grant to the author by Sun Microsystems.

## 6 References

- [1] P. S. Canning, W. R. Cook, W. L. Hill, and W. G. Olthoff. "Interfaces for strongly-typed object-oriented programming," *Proc. OOPSLA*, October 1989.
- [2] L. Cardelli and P. Wegner. "On understanding types, data abstraction, and polymorphism," *Computing Surveys*, vol. 17, p. 471, 1985.
- [3] C. Chambers and D. Ungar. "Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented language," *Proc. SIGPLAN*, June, 1989.
- [4] C. Chambers, D. Ungar, and E. Lee. "An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes," *Proc. OOPSLA*, October 1989.
- [5] M. Cline and D. Lea. *Clearer, safer, and faster C++ through annotations*. Unpublished manuscript, 1989.
- [6] W. R. Cook. "A Proposal for making Eiffel type-safe," *Proc. ECOOP*, July, 1989.
- [7] J. Guttag. "Abstract data types and the development of data structures," *Communications of the ACM*, vol 20, p576, 1989.
- [8] D. Lea. "Some storage management techniques for container classes," *The C++ Report*, January, 1990.
- [9] O. Madsen and B. Moller-Pedersen. "Virtual classes: a powerful mechanism in object-oriented programming," *Proc. OOPSLA*, October 1989.
- [10] R. Raj and H. Levy. "A compositional model for software reuse," *Proc. ECOOP*, July 1989.
- [11] B. Stroustrup. "Parameterized types for C++," *Proc. Usenix C++ Conference*, October 1988.

# Does C++ Really Need Multiple Inheritance?

T.A. Cargill\*

## Abstract

Multiple inheritance was introduced to C++ [BS86] in version 2.0 of the Cfront implementation [AT&T89]. The potential benefits of multiple inheritance (MI) must be weighed against the increased language complexity and implementation overhead incurred. In search of these benefits I have examined several published examples of MI in C++ [W&P89, SBL89, D&S89, JES89]. In all but one case, the use of MI is not essential to the constructed program; single inheritance (SI) is adequate, and (subjectively) yields simpler code. The easiest way to eliminate MI is to replace inheritance by embedding an object rather than inheriting from its class. For the example that does not succumb to this technique the SI version involves the duplication of a little code. This one candidate for legitimate use of MI is a complex piece of code, the product of long design exercise by some of the most experienced C++ programmers; it is optimistic to expect such efforts to be applied to most programs. If the legitimate use of MI remains the exception, the conclusion must be that its inclusion in C++ is of doubtful utility.

## 0. Introduction

Originally, C++ had single inheritance (SI) – a derived class could have only one base class; an inheritance hierarchy was a tree. Now, C++ 2.0 supports multiple inheritance (MI) – the number of base classes is unlimited; a hierarchy may be a directed acyclic graph (DAG). Under MI derived classes inherit all the members of all their base classes. The potential ambiguity arising from an identifier's use in more than one base class is resolved at compile time. A more subtle question arises when an ancestor base class can be reached by more than one path through the DAG: should there be a unique shared instance of the base class or a distinct copy along each path? This is addressed by a new inheritance mechanism: the virtual base class. Virtual ancestor base classes are shared; non-virtual ones are distinct..

## 1. The Cost of Multiple Inheritance

If there were no costs associated with MI, it would not much matter whether it is included in C++ or not. However, there are costs in several areas:

- complexity of the language
- complexity of the implementation
- run-time performance.

The complexity of the language can be seen in Stroustrup's discussion of ambiguities under MI in [BS89]. The behavior of virtual base classes is so counter-intuitive that Stroustrup comments the code with exclamation marks.

Pointers to member functions are an example of complexity in the implementation of MI. For quite subtle reasons [MB89], a pointer to member function cannot be implemented as machine pointers or even as pointers to dope vectors, but must be represented by a 3-tuple value: <pointer-to-function, virtual-index, object-offset>. Numerous iterations were required to

---

\*Authors address: Box 69, Louisville, Colorado 80027

implement this correctly in Cfront 2.0. The interactions with other language features, such as default arguments, were particularly troublesome.

The run-time cost of MI is small, but pervasive. To invoke a virtual function under MI requires 5 to 6 memory references, where SI requires 3 to 4; the virtual function jump tables are about 50% larger[BS89]. This is not overly-burdonsome, but because any base class might be used as an MI base, the penalty is paid by programs that only use SI.

## 2. Integer Arrays

Wiener and Pinson present a short example of the use of MI to implement an integer array [W&P89]. Unfortunately, the code does not work. The class derivation mechanism is misused, though the code recovers by means of a cast that happens to work on some hardware. See [ARK89] for a more detailed critique.

The broad conclusion here is that MI is not simple to use. If it is so complicated that the authors of two books on C++ can misuse it in a one-page example, what hope is there for the rest of us?

## 3. Monitoring a Value

The MI example in Dewhurst and Stark[D&S89] is found on page 125:

```
class Dial {
    // implementation details
protected:
    double value;
    Dial(char*, double, double);
    ~Dial();
};

class Sampler {
    // implementation details
protected:
    double freq;
virtual void sample();
    Sampler(double);
    ~Sampler();
};

class Monitor : public Sampler, public Dial {
    void sample() { value = get_value(); }
protected:
virtual double get_value();
    Monitor(char *lab, double l, double h, double f)
        : Dial(lab, l, h), Sampler(f) {}
};
```

An instance of class Dial monitors Dial::value and displays it "continuously" on a screen. The arguments to the constructor are a label and value range. An instance of class Sampler invokes sample() every freq seconds. Monitor redefines Sampler::sample(), so every freq seconds it is Monitor::sample() that is called from the Sampler component of Monitor. Monitor::sample() in turn calls Monitor's virtual function get\_value() and copies the

result to `Dial::value`, in the `Dial` component of `Monitor`. To make use of `Monitor`, another class that defines a particular `get_value()` must be derived (under SI) from `Monitor`.

However, `Monitor` can be written without MI, as follows:

```
class Dial {
    // implementation details
public:
    double value;
    Dial(char*, double, double);
    ~Dial();
};

class Monitor : public Sampler {
    Dial dial;
    void sample() { dial.value = get_value(); }
protected:
    virtual double get_value();
    Monitor(char*, double, double, int);
};
```

`Dial` has been changed to make public the part that was originally protected. `Monitor` can now embed an instance of `Dial` instead of inheriting from `Dial`. Embedding `Dial` works because the only use `Monitor` made of `Dial` was to write `Dial::value`, an operation that does not depend on inheritance. The same procedure could not be applied to `Sampler`, as it stands. `Monitor` interacts with `Sampler` by redefining a virtual function, which does depend on inheritance.

The technique of embedding an instance of an object instead of inheriting from its class will be used several times below to eliminate MI.

#### 4. Zoo Animal Taxonomy

Lippman [SLB89] presents multiple inheritance in an example on page 304, based on an information system for zoo visitors. Rather than a complete program or class implementation, there is a sequence of code fragments that demonstrate inheritance in general. Class `Panda` is derived from classes `Bear`, `Endangered` and `Herbivore`:

```
class Endangered {
public:
    highlight(short);
};

class Herbivore {
public:
    highlight(short);
};

class Panda : public Bear, public Endangered, public Herbivore
{
public:
    locate();
};
```



An attempt to call `highlight()` from within `Panda::locate()` is used to show that ambiguity between the `highlight()`'s in `Endangered` and `Herbivore` results in a compile-time error. To correct this, `Panda::highlight()` is introduced to call each `highlight()` in turn, unambiguously:

```
Panda::highlight()
{
    Endangered::highlight();
    Herbivore::highlight();
}
```

The fact that class `Panda` inherits from `Endangered` and `Herbivore` is not exploited. All that matters is that `Panda::highlight()` calls their respective `highlight()` functions. Using the same procedure as above, the effect can be obtained by embedding objects instead of inheriting from their class:

```
class Panda : public Bear {
    Endangered endangered;
    Herbivore herbivore;
public:
    locate();
    highlight();
};

Panda::highlight()
{
    endangered.highlight();
    herbivore.highlight();
}
```

MI appears again on page 346 of Lippman, but the example is more concerned with illustrating details than showing an application of MI.

## 5. The Iostream Library

Shapiro [JES89] describes how multiple inheritance is used in the `Iostream` library [AT&T89]. The code is a simplified version of `Iostream` that uses MI in the same way as `Iostream`. The example is deceptively short; it is perhaps the densest one hundred lines of C++ published to date. It uses MI twice, for a different reason in each case.

The first use of MI takes the union of the operators defined for an input class and those defined for an output class to construct an input/output class. To concentrate on the essential details, the code presented here is further simplified, but its essential properties are maintained.

```
class streambuf {
    friend istream;
    friend ostream;
protected:
    // dummy function bodies
    int sbumpc() { return 0; }
    void sputc(int) {}
};
```

```

class ios {
public:
protected:
    streambuf *bp;    // initialization code omitted
};

class ostream : public virtual ios {
public:
    ostream&operator<<(char c){
        bp->sputc(c);
        return *this;
    }
};

class istream : public virtual ios {
public:
    istream&operator>>(char &c) {
        c = bp->sgetc();
        return *this;
    }
};

class iostream : public istream, public ostream {};

```

Class `streambuf` implements both input and output primitives, using buffering code not shown here. Class `ios` holds a pointer to a `streambuf`, used by derived classes. [The details of how `streambuf::bp` is initialized by various omitted constructors is not essential to understanding the code.] The public interface of class `istream` is `operator>>`, which calls `streambuf::sgetc()`. Only the input part of `streambuf` is accessible from `istream`. This provides a compile-time check that only input operations are applied. Similarly, `ostream::operator<<` calls `streambuf::sputc()`.

Class `iostream` re-combines the input and output halves by inheriting from both `istream` and `ostream`. Because `ios` is a virtual base class of both, `iostream` contains a unique `ios`, and therefore refers to a unique `streambuf`. Nothing new is added by `iostream`; it simply approximates the union of `istream` and `ostream`. It is not exactly the union of `istream` and `ostream`. Although, both `operator>>` and `operator<<` are members of `iostream`, their use cannot be mixed in a single expression. If `io` is an instance of `iostream`, the following is invalid:

```

char a, b;
io << a >> b;

```

This expression is parsed as `(io<<a)>>b`, and the type of the sub-expression `(io<<a)` is `ostream&`. `Operator>>` cannot be applied to an `ostream`; only output can be performed on an `ostream`. In general, for any expression using an instance of an `iostream` the first operator causes `iostream`'s input/output duality to collapse to exclusively input or output for that expression. [The complete quantum model of C++ is beyond the scope of this paper!]

Since an `iostream` must commit itself to being exclusively an `istream` or an `ostream` whenever it is used, there is an alternative SI implementation. Once again MI can be replaced by embedding:

```

class iostream_SI {
public:

```

```

        istream    in;
        ostream    out;
    };

```

If `io_SI` is an instance of `iostream_SI`, the following expressions would be used for input and output, respectively:

```

io_SI.in >> a >> b;
io_SI.out << a << b;

```

Since both `istream` and `ostream` inherit from `ios`, `io_SI` carries two copies of `ios::bp`. But one extra pointer per input/output stream is negligible compared to the other implicit costs for a stream. Of course, both pointers point to the same `streambuf`, so the semantics are preserved.

There is a second SI solution; it does not use the embedding technique:

```

class streambuf {
friend ios;
protected:
    int    sbumpc()    { return 0; }
    void    sputc(int) {}
};

class ios {
protected:
    streambuf *bp;
public:
    ios &operator<<(char c) {
        bp->sputc(c);
        return *this;
    }
    ios &operator>>(char &c) {
        c = bp->sbumpc();
        return *this;
    }
};

class istream : private ios {
public:
    ios::operator>>;
};

class ostream : private ios {
public:
    ios::operator<<;
};

```

In this case both input and output operators are members of the base class, `ios`. Now `istream` and `ostream` must be restrictions of class `ios`, limited to input and output operations, respectively. We get exactly that if `istream` and `ostream` take `ios` as a private base class and then promote to public only the operator that each needs.

This solution has one less level of inheritance and one less friend class. Using the access control mechanisms in the language for controlling access is perhaps more natural than using MI. Since the operators in `ios` return `ios&`'s, input and output operators can be mixed in an expression. Unfortunately, we now have the compliment of the problem found with the original MI `iostream`: it can no longer be detected at compile-time if an instance of `istream` attempts output, or an `ostream` attempts input, as long as the first of a sequence of operators is correct.

Shopiro's second use of MI is in the implementation side of the package. Class `streambuf` is an abstract base class; a derived class must implement its buffer reads and writes for a particular device. Class `filebuf` does it for files:

```
class filebuf : public streambuf {
    friend fstreambase;
    friend fstream;
    friend ifstream;
    friend ostream;
    int fd;
    void seek(long, int);
};

class fstreambase : virtual public ios {
public:
    void seek(long o, int w) {
        ((filebuf*)bp)->seek(o, w);    // cast
    }
};

class ifstream : public fstreambase, public istream {};
class ostream : public fstreambase, public ostream {};
class fstream : public fstreambase, public iostream {};
```

Classes `ifstream`, `ostream` and `fstream` are specializations of `istream`, `ostream` and `iostream` that have a `seek()` member function inherited from class `fstreambase`. Note the presence of an unsightly cast in `fstreambase::seek()`; `ios::bp` is an `streambuf*` but `bp` is initialized with a `filebuf*`, by the various constructors not shown here. Removal of this cast, so that everything is properly type-checked, is possible, but beyond the scope of this paper.

The SI version of this code leaves `filebuf` unaltered, but eliminates `fstreambase`:

```
class ifstream_SI : public istream {
public:
    void seek(long o, int w) {
        ((filebuf*)bp)->seek(o, w);
    }
};

class ostream_SI : public ostream {
public:
    void seek(long o, int w) {
        ((filebuf*)bp)->seek(o, w);
    }
};
```

```

    }
};

class fstream_SI {
public:
    ofstream_SI in;
    ifstream_SI out;
};

```

Class `fstream_SI` embeds `ofstream_SI` and `ifstream_SI`, in the same way as `iostream_SI` embeds `istream` and `ostream`. A larger difference is that `fstreambase` has been eliminated at the expense of `ifstream` and `ofstream` having to have their own implementations of `seek()`. It is unquestionably better to have a single implementation of `seek()`. But that goal has to be balanced against the overall complexity of the MI solution. For example, under MI `ios` is inherited along three paths from `fstream`, viz.:

```

fstream : fstreambase : ios
fstream : iostream : istream : ios
fstream : iostream : ostream : ios

```

This is a needlessly complicated software architecture to address the relatively well understood problem of reading and writing files. Unfortunately, this is the only published material to recommend to a C++ novice looking for an example of how to use MI.

## 6. Libraries

Another case for MI is that the multiple base classes may have been designed in isolation and supplied as free-standing libraries [BS89]. In this case the programmer may not be able to use the techniques suggested above. This argument appears to be sound, but no account of MI addressing this problem has been published.

## 7. Conclusion

In the majority of published examples, multiple inheritance in C++ is not required to express the programs. In most cases a single inheritance solution is simpler – inheritance from a class being replaced by the embedding an object. In light of this we should view multiple inheritance in C++ as a large scale research project, in which we must evaluate its usefulness and weigh its costs. The results to date suggest that multiple inheritance is not paying its way.

## 8. References

- [BS86] B. Stroustrup *The C++ Programming Language*, Addison-Wesley, 1986.
- [BS89] B. Stroustrup The Evolution of C++: 1985 to 1989, *Computing Systems* 2:3, Summer 1989.
- [D&S89] S.C. Dewhurst, K.T. Stark *Programming in C++*, Prentice Hall, 1989.
- [SBL89] S.B. Lippman *C++ Primer*, Addison-Wesley, 1989.
- [JES89] J.E. Shopiro An Example of Multiple Inheritance in C++: A Model of the `iostream` Library, *Sigplan Notices* 24:12, December 1989.



- [W&P89] R.S. Wiener, L.J. Pinson, A Practical Example of Multiple Inheritance in C++, *Sigplan Notices* 24:9, September 1989.
- [ARK89] A.R. Koenig et al. Letter, *Sigplan Notices* 24:12, December 1989.
- [MB89] M. Ball, Personal communication, 1989.
- [AT&T89] *UNIX System V AT&T C++ Language System Release 2.0 Manual*, 1989



## *The USENIX Association*

*T*he USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

*Computing Systems*, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association  
2560 Ninth Street, Suite 215  
Berkeley, CA 94710

Telephone: 415 528-8649  
Email: [office@usenix.org](mailto:office@usenix.org)

